

Sistema de Banco de Dados

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

Tradução da 5^a edição

Revisão Técnica

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

Tradução

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



Do original:

Database system concepts

Tradução autorizada do idioma inglês da edição publicada por McGraw-Hill – McGraw-Hill Companies, Inc.

Copyright © 2006 by Mc-Graw-Hill Companies, Inc.

© 2006, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998.

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

Editoração Eletrônica

Estúdio Castellani

Revisão Gráfica

Marco Antonio Correa

Marilia Pinto de Oliveira

Copidesque

Adriana Kramer

Projeto Gráfico

Elsevier Editora Ltda.

A Qualidade da Informação.

Rua Sete de Setembro, 111 – 16º andar

20050-006 Rio de Janeiro RJ Brasil

Telefone: (21) 3970-9300 FAX: (21) 2507-1991

E-mail: info@elsevier.com.br

Escritório São Paulo:

Rua Quintana, 753/8º andar

04569-011 Brooklin São Paulo SP

Tel.: (11) 5105-8555

ISBN 13: 978-85-352-1107-8

ISBN 10: 85-352-1107-8

Edição original: 0-07-295886-3

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

Central de atendimento

Tel: 0800-265340

Rua Sete de Setembro, 111, 16º andar – Centro – Rio de Janeiro

e-mail: info@elsevier.com.br

site: www.campus.com.br

CIP-Brasil. Catalogação-na-fonte.

Sindicato Nacional dos Editores de Livros, RJ

S576a

Silberschatz, Abraham

Sistema de banco de dados / Abraham Silberschatz,

Henry F. Korth, S. Sudarshan ; tradução de Daniel Vieira.

– Rio de Janeiro : Elsevier, 2006.

II.

Tradução de: *Database system concepts*, 5th ed

Apêndice

Inclui bibliografia

ISBN 85-352-1107-8

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

06-1033

CDD 005.74

CDU 004.65

Indexação e hashing

Muitas consultas referenciam apenas uma pequena proporção dos registros em um arquivo. Por exemplo, uma consulta como "Encontrar todas as contas na agência Perryridge" ou "Encontrar o saldo do número de conta A-101" referencia apenas uma fração dos registros de conta. Não é eficiente para o sistema ler cada registro e verificar o campo *nome_agência* para encontrar o nome "Perryridge" ou verificar o campo *número-conta* para encontrar o valor A-101. O ideal é que o sistema seja capaz de localizar esses registros diretamente. Para permitir essas formas de acesso, projetamos estruturas especiais que associamos aos arquivos.

Conceitos básicos

Um índice para um arquivo em um sistema de banco de dados funciona quase da mesma forma que o índice deste livro. Se quisermos descobrir algo sobre um assunto em particular (especificado por uma palavra ou uma frase) neste livro, podemos procurar o assunto no índice ao final do livro, encontrar a página em que ele ocorre e depois ler as páginas para encontrar as informações que estamos procurando. As palavras no índice estão em ordem, facilitando a localização daquela que estamos procurando. Além do mais, o índice é muito menor do que o livro, reduzindo ainda mais o esforço necessário para encontrar as palavras que estamos procurando.

Os índices no sistema de banco de dados desempenham o mesmo papel dos índices de livro nas bibliotecas. Por exemplo, para apanhar um registro de *conta* dado o número da conta, o sistema de banco de dados pesquisaria um índice para descobrir em que bloco do disco o registro correspondente reside, e depois apanharia o bloco do disco, para obter o registro de *conta*.

Manter uma lista classificada de números de conta não funcionaria bem em muitos sistemas grandes de banco de dados, com milhões de contas, pois o próprio índice seria muito grande; além disso, embora manter um índice classificado reduza o tempo de busca, encontrar uma conta ainda pode ser um tanto demorado. Em vez disso, técnicas de indexação mais sofisticadas podem ser utilizadas. Discutiremos várias dessas técnicas neste capítulo.

Existem dois tipos básicos de índices:

- **Índices ordenados.** Baseados em uma ordem classificada dos valores.
- **Índices de hash.** Baseados em uma distribuição uniforme de valores por um intervalo de buckets (baldes). O bucket ao qual um valor é atribuído é determinado por uma função, chamada *função de hash*.

Vamos considerar várias técnicas para a indexação ordenada e o hashing. Nenhuma técnica é a melhor. Em vez disso, cada técnica é mais adequada a determinadas aplicações de banco de dados. Cada técnica precisa ser avaliada com base nestes fatores:

- **Tipos de acesso:** os tipos de acesso que são aceitos com eficiência. Os tipos de acesso podem incluir a localização de registros com um valor de atributo especificado e a localização de registros cujos valores de atributos se encontram em um intervalo especificado.
- **Tempo de acesso:** o tempo gasto para encontrar determinado item de dados, ou conjunto de itens, usando a técnica em questão.
- **Tempo de inserção:** o tempo gasto para inserir um novo item de dados. Esse valor inclui o tempo gasto

para encontrar o local atual para inserir o novo item de dados, além do tempo gasto para atualizar a estrutura de índice.

- **Tempo de exclusão:** o tempo gasto para excluir um item de dados. Esse valor inclui o tempo gasto para localizar o item a ser excluído, além do tempo gasto para atualizar a estrutura de índice.
- **Espaço adicional:** o espaço adicional ocupado por uma estrutura de índice. Desde que a quantidade de espaço adicional seja moderada, normalmente vale a pena sacrificar o espaço para conseguir um desempenho melhor.

Normalmente, queremos ter mais de um índice para um arquivo. Por exemplo, podemos querer pesquisar um livro por autor, por assunto ou por título.

Um atributo ou conjunto de atributos utilizados para pesquisar registros em um arquivo é denominado **chave de busca**. Observe que essa definição de *chave* difere daquela usada na *chave primária*, *chave candidata* e *superchave*. Esse duplo significado para *chave* (infelizmente) está bem estabelecido na prática. Usando nossa noção de chave de busca, vemos que, se existirem vários índices em um arquivo, várias chaves de busca serão usadas.

Índices ordenados

Para obter um acesso aleatório rápido aos registros em um arquivo, podemos usar uma estrutura de índice. Cada estrutura de índice está associada a uma determinada chave de busca. Assim como o índice de um livro ou um catálogo da biblioteca, um índice ordenado armazena os valores das chaves de busca em ordem classificada e associa a cada chave de busca os registros que a contêm.

Os próprios registros no arquivo indexado podem ser armazenados em alguma ordem classificada, assim como os livros em uma biblioteca são armazenados de acordo com algum atributo, como o número decimal de Dewey. Um ar-

quivo pode ter vários índices, em diferentes chaves de busca. Se o arquivo contendo os registros for ordenado sequencialmente, um índice de agrupamento é um índice cuja chave de busca também define a ordem seqüencial do arquivo. Os índices de agrupamento também são chamados **índices primários**; o termo índice primário parece indicar um índice sobre uma chave primária, mas esses índices na verdade podem ser montados com base em qualquer chave de busca. A chave de busca de um índice de agrupamento normalmente é a chave primária, embora isso não seja necessariamente dessa forma. Os índices cuja chave de busca especifica uma ordem diferente da ordem seqüencial do arquivo são chamados **índices não de agrupamento**, ou **índices secundários**. Os termos “agrupado” e “não agrupado” normalmente são usados no lugar de “agrupamento” e “não de agrupamento”.

Nas seções “Índices densos e esparsos” a “Atualização do índice”, consideramos que todos os arquivos são ordenados seqüencialmente em alguma chave de busca. Tais arquivos, com um índice agrupado sobre a chave de busca, são chamados **arquivos seqüenciais indexados**. Eles representam um dos esquemas de índice mais antigos utilizados nos sistemas de banco de dados. Eles são projetados para aplicações que exigem tanto o processamento seqüencial do arquivo inteiro quanto o acesso aleatório a registros individuais. Na seção “Índices secundários”, explicamos os índices secundários.

A Figura 12.1 mostra um arquivo seqüencial de registros de *conta* apanhados do nosso exemplo bancário. No exemplo da Figura 12.1, os registros são armazenados na ordem da chave de busca, com *nome_agência* sendo usado como chave de busca.

Índices densos e esparsos

Um **registro de índice**, ou **entrada de índice**, consiste em um valor da chave de busca e ponteiros para um ou mais re-

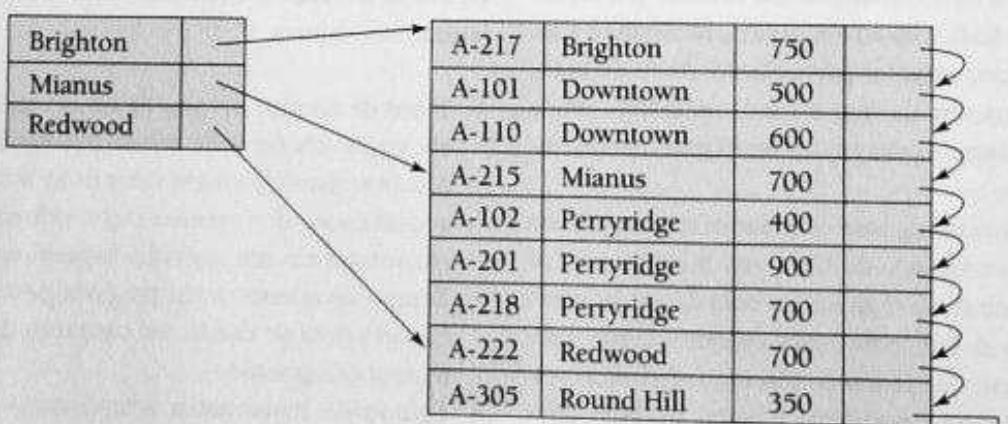


Figura 12.1 Arquivo seqüencial para registros de conta.

gistros com esse valor como seu valor de chave de busca. O ponteiro para um registro consiste no identificador de um bloco do disco e um deslocamento dentro do bloco de disco, para identificar o registro dentro do bloco.

Existem dois tipos de índices ordenados que podemos utilizar:

- **Índice denso:** um registro de índice aparece para cada valor de chave de busca no arquivo. Em um índice de agrupamento denso, o registro de índice contém o valor da chave de busca e um ponteiro para o primeiro registro de dados com esse valor da chave de busca. O restante dos registros com o mesmo valor da chave de busca seria armazenado sequencialmente após o primeiro registro, pois, como o índice é agrupado, os registros são classificados sobre a mesma chave de busca.

Implementações de índice denso podem armazenar uma lista de ponteiros para todos os registros com o mesmo valor de chave de busca; isso não é essencial para os índices agrupados.

- **Índice esparsos:** um registro de índice aparece para somente alguns dos valores da chave de busca. Como acontece nos índices densos, cada registro de índice contém um valor de chave de busca e um ponteiro para o primeiro registro de dados com esse valor de chave de busca. Para localizar um registro, encontramos a entrada de índice com o maior valor de chave de busca que é menor ou igual ao valor de chave de busca pelo qual estamos procurando. Começamos no registro apontado pela entrada de índice e acompanhamos os ponteiros no arquivo até encontrarmos o registro desejado.

As Figuras 12.2 e 12.3 mostram os índices densos e esparsos, respectivamente, para o arquivo de *conta*. Suponha que estejamos procurando registros para a agência Perryridge. Usando o índice denso da Figura 12.2, acompanhamos o ponteiro diretamente para o primeiro registro de Perryridge. Processamos esse registro e acompanhamos o

ponteiro nesse registro para localizar o próximo registro na ordem da chave de busca (*nome_agência*). Continuamos processando os registros até encontrarmos um registro para uma agência diferente de Perryridge. Se estivermos usando o índice esparsos (Figura 12.3), não encontraremos uma entrada de índice para "Perryridge". Como a última entrada (em ordem alfabética) antes de "Perryridge" é "Mianus", acompanhamos esse ponteiro. Depois, lemos o arquivo *conta* em ordem sequencial até encontrarmos o primeiro registro Perryridge, e começamos a processar nesse ponto.

Como vimos, geralmente é mais rápido localizar um registro se tivermos um índice denso ao invés de um índice esparsos. Porém, os índices esparsos possuem vantagens em relação aos índices densos, pois exigem menos espaço e impõem menos sobrecarga de manutenção para inserções e exclusões.

Existe uma escolha que o projetista do sistema deve fazer entre tempo de acesso e espaço adicional. Embora a decisão com relação a isso dependa da aplicação específica, um bom meio-termo é usar um índice esparsos com uma entrada de índice por bloco. O motivo para esse esquema ser uma boa escolha é que o custo dominante no processamento de uma solicitação de banco de dados é o tempo gasto para trazer um bloco do disco para a memória principal. Depois que o bloco for trazido, o tempo para varrer o bloco inteiro é mínimo. Usando esse índice esparsos, localizamos o bloco que contém o registro que estamos buscando. Assim, a menos que o registro esteja em um bloco de estouro (ver seção "Organização sequencial de arquivos" do Capítulo 11), reduzimos os acessos ao bloco enquanto mantemos o tamanho do índice (portanto, nossa sobrecarga de espaço) o menor possível.

Para que a técnica anterior seja totalmente genérica, temos de considerar o caso em que os registros para um valor de chave de busca ocupam vários blocos. É fácil modificar nosso esquema para lidar com essa situação.

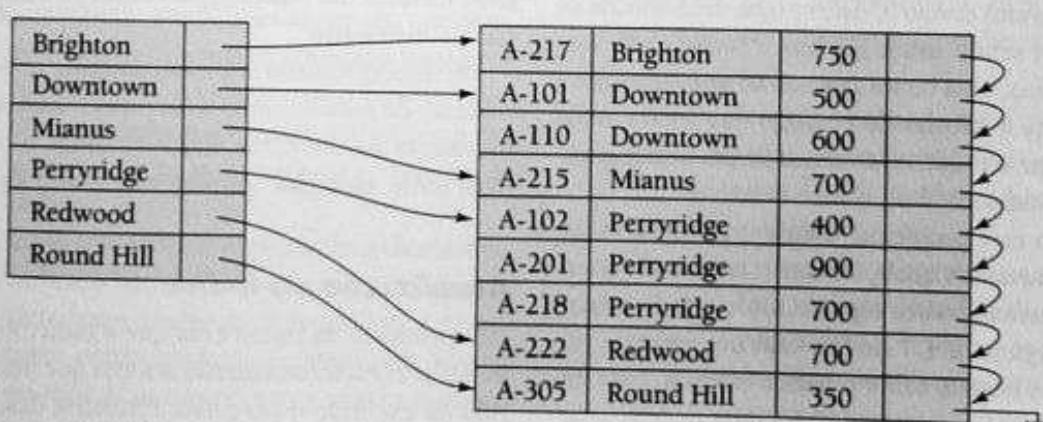
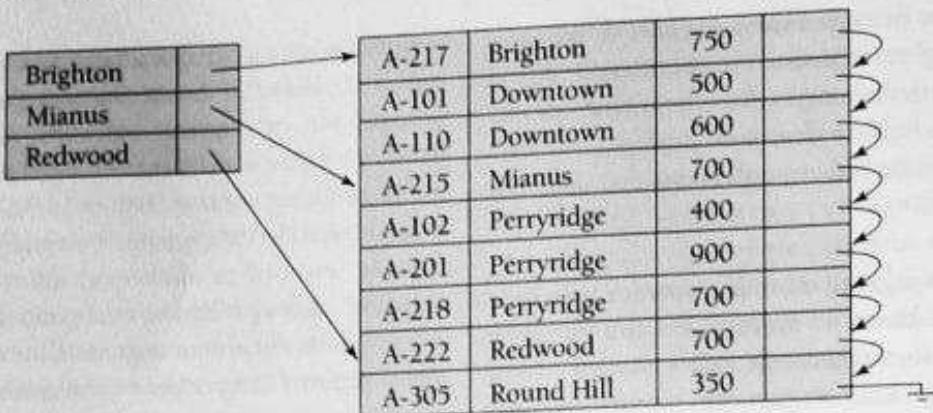


Figura 12.2 Índice denso.

**Figura 12.3** Índice esparsos.

Índices multiníveis

Mesmo se usarmos um índice esparsos, o próprio índice pode se tornar muito grande para um processamento eficiente. Na prática, não é razoável ter um arquivo com 100.000 registros, com 10 registros armazenados em cada bloco. Se tivermos um registro de índice por bloco, o índice terá 10.000 registros. Os registros de índice são menores do que os registros de dados, de modo que vamos considerar que 100 registros de índice cabem em um bloco. Assim, nosso índice ocupa 100 blocos. Esses grandes índices são armazenados como arquivos sequenciais no disco.

Se um índice for suficientemente pequeno para ser mantido na memória principal, o tempo de busca para encontrar uma entrada será baixo. Porém, se o índice for tão grande que deva ser mantido no disco, uma busca por uma entrada exige várias leituras de bloco de disco. A busca binária pode ser usada sobre o arquivo de índice para localizar uma entrada, mas a busca ainda tem um grande custo. Se o índice ocupa b blocos, a busca binária exige até $\lceil \log_2(b) \rceil$ blocos a serem lidos. ($\lceil x \rceil$ indica o menor inteiro que seja maior ou igual a x , ou seja, arredondamos para cima.) Para o nosso índice de 100 blocos, a busca binária exige sete leituras de bloco. Em um sistema de disco em que uma leitura de bloco exige 30 milissegundos, a busca usará 210 milissegundos, o que é muito tempo. Observe que, se os blocos de estouro estiverem sendo usados, a busca binária não será possível. Nesse caso, uma busca sequencial normalmente é usada, e isso exige b leituras de bloco, o que levará ainda mais tempo. Assim, o processo de busca de um índice grande pode ser dispendioso.

Para lidar com esse problema, tratamos o índice exatamente como trataríamos qualquer outro arquivo sequencial, e construímos um índice esparsos sobre o índice agrupado, como na Figura 12.4. Para localizar um registro, primeiro use a busca binária sobre o índice externo, para encontrar o registro para o maior valor de chave de busca menor ou igual a um que desejamos. O ponteiro aponta para

um bloco do índice interno. Varremos esse bloco até encontrar o registro que tem o maior valor de chave de busca menor ou igual ao que desejamos. O ponteiro nesse registro aponta para o bloco do arquivo que contém o registro pelo qual estamos procurando.

Usando os dois níveis de indexação, lemos apenas um bloco de índice, em vez dos sete que lemos com a busca binária, se considerarmos que o índice externo já está na memória principal. Se o nosso arquivo for extremamente grande, até mesmo o índice externo pode crescer muito para caber na memória principal. Nesse caso, podemos criar outro nível de índice. Na verdade, podemos repetir esse processo tantas vezes quantas forem necessárias. Os índices com dois ou mais níveis são chamados de índices multiníveis. A busca de registros com um índice multinível exige muito menos operações de E/S do que a busca binária de registros. Cada nível de índice poderia corresponder a uma unidade de armazenamento físico. Assim, podemos ter índices nos níveis de trilha, cilindro e disco.

Um dicionário típico é um exemplo de um índice multinível em outro mundo, que não o de banco de dados. O cabeçalho de cada página lista a primeira palavra alfabeticamente nessa página. Esse tipo de índice de livro é um índice multinível: as palavras no alto de cada página do índice do livro formam um índice esparsos sobre o conteúdo das páginas do dicionário.

Os índices multiníveis estão bastante relacionados às estruturas de árvore, como as árvores binárias usadas para a indexação na memória. Examinaremos o relacionamento mais tarde, na seção "Arquivos de índice de árvore B".

Atualização do índice

Independentemente da forma com que o índice é utilizado, cada índice precisa ser atualizado sempre que um registro é inserido ou excluído do arquivo. Primeiro, descrevemos os algoritmos para atualização de índices de único nível.

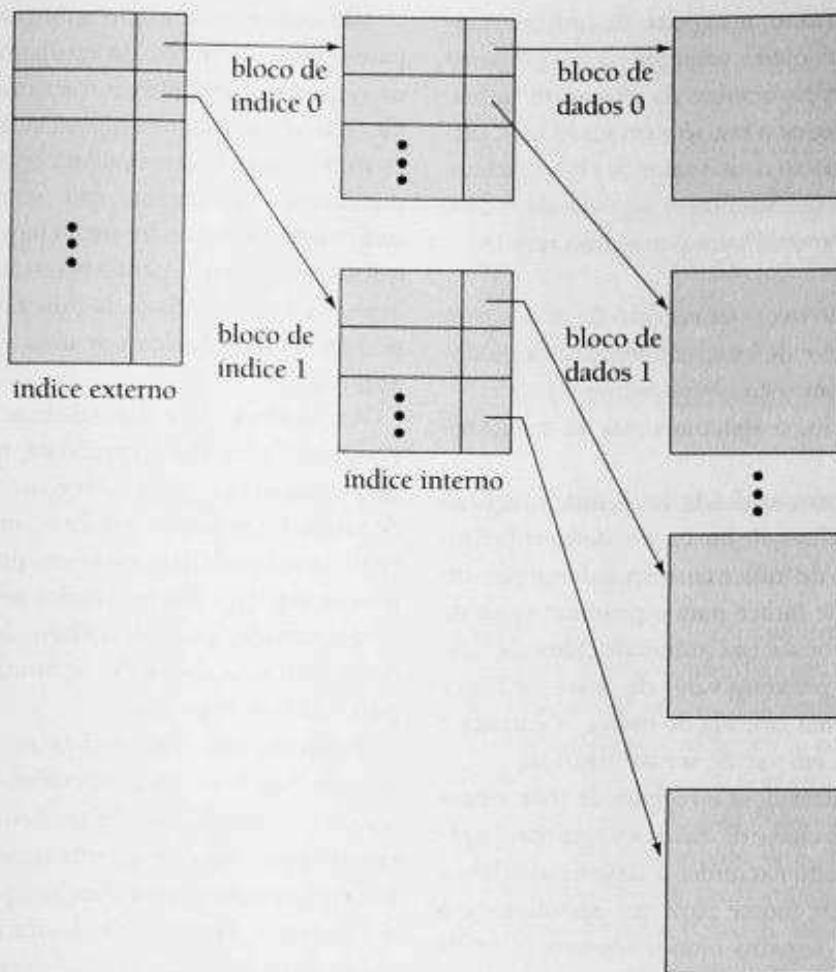


Figura 12.4 índice esparsos de dois níveis.

- **Inserção.** Primeiro, o sistema realiza uma pesquisa usando o valor de chave de busca que aparece no registro a ser inserido. As ações que o sistema realiza em seguida dependem se o índice é denso ou esparsos:

- Índices densos:

1. Se o valor da chave de busca não aparece no índice, o sistema insere um registro de índice com o valor de chave de busca no índice, na posição apropriada.
2. Caso contrário, a seguintes ações são tomadas:
 - a. Se o registro de índice armazena ponteiros para todos os registros com o mesmo valor de chave de busca, o sistema acrescenta um ponteiro para o novo registro no registro de índice.
 - b. Caso contrário, o registro de índice armazena um ponteiro somente no primeiro registro com o valor da chave de busca. O sistema, então, coloca o registro sendo inserido após os outros registros com os mesmos valores de chave de busca.

- Índices esparsos: Consideramos que o índice armazena uma entrada para cada bloco. Se o sistema cria um novo bloco, ele insere no índice o primeiro valor de chave de busca (na ordem de chave de busca) que aparece no novo bloco. Por outro lado, se o novo registro tiver o menor valor de chave de busca em seu bloco, o sistema atualiza a entrada de índice apontando para o bloco; se não, o sistema não faz qualquer mudança no índice.

- **Exclusão.** Para excluir um registro, o sistema primeiro pesquisa o registro a ser excluído. As ações que o sistema toma em seguida dependem se o índice é denso ou esparsos.

- Índices densos:

1. Se o registro excluído foi o único registro com seu valor específico de chave de busca, então o sistema retira do índice o registro de índice correspondente.
2. Caso contrário, as seguintes ações são tomadas:
 - a. Se o registro de índice armazena ponteiros para todos os registros com o mesmo valor de chave de índice, o sistema exclui do registro de índice o ponteiro para o registro excluído.

- b. Caso contrário, o registro de índice armazena um ponteiro somente para o primeiro registro com o valor da chave de busca. Nesse caso, se o registro excluído foi o primeiro registro com o valor da chave de busca, o sistema atualiza o registro de índice para que aponte para o próximo registro.

□ Índices esparsos:

1. Se o índice não tiver um registro de índice com o valor de chave de busca do registro excluído, nada precisa ser feito com o índice.
2. Caso contrário, o sistema toma as seguintes ações:
 - a. Se o registro excluído foi o único registro com sua chave de busca, o sistema substitui o registro de índice correspondente por um registro de índice para o próximo valor da chave de busca (na ordem de chave de busca). Se o próximo valor de chave de busca já tiver uma entrada de índice, a entrada é excluída, em vez de ser substituída.
 - b. Caso contrário, se o registro de índice para o valor da chave de busca apontar para o registro sendo excluído, o sistema atualiza o registro de índice para que aponte para o próximo registro com o mesmo valor da chave de busca.

Os algoritmos de inserção e exclusão para índices multível são uma extensão simples do esquema que acabamos de descrever. Na exclusão ou inserção, o sistema atualiza o índice de nível mais baixo conforme descrevemos. Com relação ao segundo nível, o índice de menor nível é simplesmente um arquivo contendo registros – assim, se houver qualquer mudança no índice de menor nível, o sistema atualiza o índice de segundo nível conforme descrevemos. A mesma técnica se aplica a outros níveis do índice, se houver algum.

Índices secundários

Os índices secundários precisam ser densos, com uma entrada de índice para cada valor de chave de busca, e um ponteiro para cada registro no arquivo. Um índice agrupado pode ser esparsos, armazenando apenas alguns dos valores da chave de busca, pois sempre é possível encontrar registros com valores intermediários da chave de busca por um acesso seqüencial a uma parte do arquivo, como já descrevemos. Se um índice secundário armazenar apenas algumas das chaves de busca, os registros com valores intermediários de chave de busca podem estar em qualquer lugar no arquivo e, em geral, não podemos encontrá-los sem pesquisar o arquivo inteiro.

Um índice secundário sobre uma chave candidata se parece com um índice de agrupamento denso, exceto que os registros apontados por valores sucessivos no índice não são armazenados seqüencialmente. Porém, em geral, os índices secundários podem ter uma estrutura diferente dos índices de agrupamento. Se a chave de busca de um índice agrupado não for uma chave candidata, é suficiente que o índice aponte para o primeiro registro com um valor específico para a chave de busca, pois os outros registros podem ser apanhados por uma varredura seqüencial do arquivo.

Ao contrário, se a chave de busca de um índice secundário não for uma chave candidata, não será suficiente apontar apenas para o primeiro registro com cada valor de chave de busca. Os registros restantes com o mesmo valor de chave de busca poderiam estar em qualquer lugar no arquivo, pois os registros são ordenados pela chave de busca do índice agrupado, e não pela chave de busca do índice secundário. Portanto, um índice secundário precisa ter ponteiros para todos os registros.

Podemos usar um nível extra de indireção para implementar índices secundários sobre chaves de busca que não são chaves candidatas. Os ponteiros nesse índice secundário não apontam diretamente para o arquivo. Em vez disso, cada um aponta para um bucket que contém ponteiros para o arquivo. A Figura 12.5 mostra a estrutura de um índice secundário que usa um nível extra de indireção para o arquivo de conta, sobre a chave de busca saldo.

Uma varredura seqüencial na ordem do índice agrupado é eficiente porque os registros no arquivo são armazenados fisicamente na mesma ordem do índice. Porém, não podemos (exceto em casos especiais raros) armazenar um arquivo fisicamente ordenado pela chave de busca do índice agrupado e pela chave de busca de um índice secundário. Como a ordem da chave secundária e a ordem da chave física são diferentes, se tentarmos varrer o arquivo seqüencialmente na ordem da chave secundária, a leitura de cada registro provavelmente exigirá a leitura de um novo bloco do disco, o que é muito lento.

O procedimento descrito anteriormente para exclusão e inserção também pode ser aplicado a índices secundários; as ações tomadas são aquelas descritas para índices densos armazenando um ponteiro para cada registro no arquivo. Se um arquivo tiver vários índices, sempre que o arquivo for modificado, cada índice terá de ser atualizado.

Os índices secundários melhoraram o desempenho das consultas que usam chaves diferentes da chave de busca do índice agrupado. Porém, eles impõem uma sobrecarga significativa na modificação do banco de dados. O projetista de um banco de dados decide quais índices secundários são desejados com base em uma estimativa da frequência relativa das consultas e modificações.

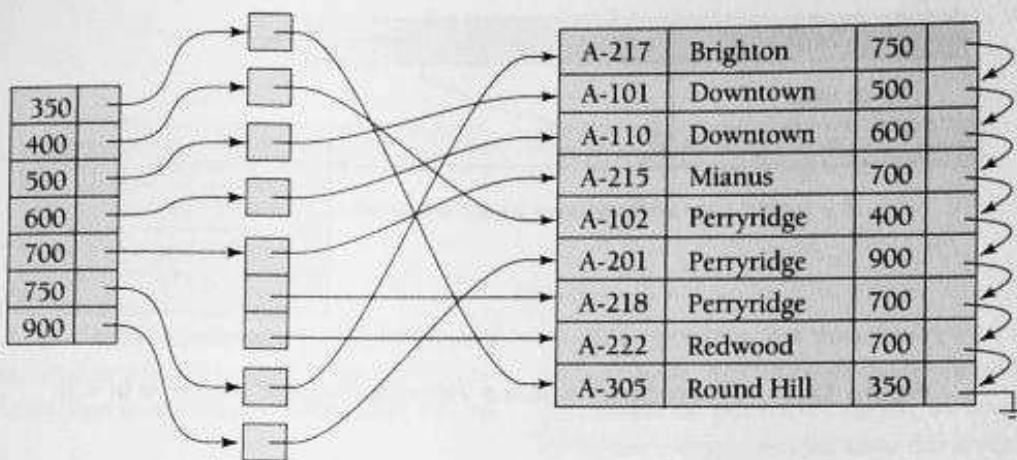


Figura 12.5 Índice secundário no arquivo de *conta*, sobre a chave não candidata *saldo*.

Arquivos de índice de árvore B⁺

A principal desvantagem da organização de arquivo sequencial indexada é que o desempenho diminui enquanto o arquivo cresce, tanto para pesquisas de índice quanto para varreduras sequenciais pelos dados. Embora essa degradação possa ser remediada pela reorganização do arquivo, as reorganizações frequentes não são desejáveis.

A estrutura de índice de árvore B⁺ é a mais utilizada das várias estruturas de índice que mantém sua eficiência apesar da inserção e exclusão de dados. Um índice de árvore B⁺ tem a forma de uma árvore balanceada, em que cada caminho da raiz da árvore até uma folha da árvore é do mesmo tamanho. Cada nó não-folha na árvore tem entre $\lceil n/2 \rceil$ e n filhos, onde n é fixo para um árvore em particular.

Veremos que a estrutura de árvore B⁺ impõe sobrecarga de desempenho na inserção e na exclusão, além de aumentar o espaço adicional. A sobrecarga é aceitável até mesmo para arquivos modificados com frequência, pois o custo da reorganização do arquivo é evitado. Além do mais, como os nós podem estar vazios até a metade (se tiverem o número mínimo de filhos), existe algum espaço desperdiçado. Esse espaço adicional também é aceitável dados os benefícios de desempenho da estrutura de árvore B⁺.

Estrutura de uma árvore B⁺

Um índice de árvore B⁺ é um índice de multível, mas tem uma estrutura que difere daquela do arquivo sequencial indexado multível. A Figura 12.6 mostra um nó típico de uma árvore B⁺. Ele contém até $n - 1$ valores de chave de

busca K_1, K_2, \dots, K_{n-1} e n ponteiros P_1, P_2, \dots, P_n . Os valores de chave de busca dentro de um nó são mantidos em ordem classificada; assim, se $i < j$, então $K_i < K_j$.

Consideramos primeiro a estrutura dos nós de folha. Para $i = 1, 2, \dots, n - 1$, o ponteiro P_i aponta para um registro de arquivo com valor de chave de busca K_i . A estrutura de bucket é usada somente se a chave de busca não formar uma chave candidata e se o arquivo não for classificado na ordem do valor da chave de busca. (Estudaremos mais tarde, na seção “Chaves de busca não exclusivas”, como evitar a criação de buckets, fazendo com que a chave de busca pareça ser exclusiva.) O ponteiro P_n tem uma finalidade especial, que discutiremos em breve.

A Figura 12.7 mostra um nó de folha de uma árvore B⁺ para o arquivo de *conta*, em que escolhemos n como sendo 3, e a chave de busca é *nome_agência*. Observe que, como o arquivo de *conta* é ordenado por *nome_agência*, os ponteiros no nó de folha apontam diretamente para o arquivo.

Agora que vimos a estrutura de um nó de folha, vamos considerar como os valores da chave de busca são atribuídos aos nós em particular. Cada folha pode manter até $n - 1$ valores. Permitimos que os nós de folha contenham pelo menos $\lceil (n - 1)/2 \rceil$ valores. Os intervalos de valores em cada folha não se sobreponem. Assim, se L_i e L_j são nós de folha e $i < j$, então cada valor de chave de busca em L_i é menor que cada valor de chave de busca em L_j . Se o índice de árvore B⁺ tiver de ser um índice denso, cada valor de chave de busca precisa aparecer em algum nó de folha.

Agora, podemos explicar o uso do ponteiro P_n . Como existe uma ordem linear nas folhas, com base nos valores de

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-----	-----------	-----------	-------

Figura 12.6 Nô típico de uma árvore B⁺.

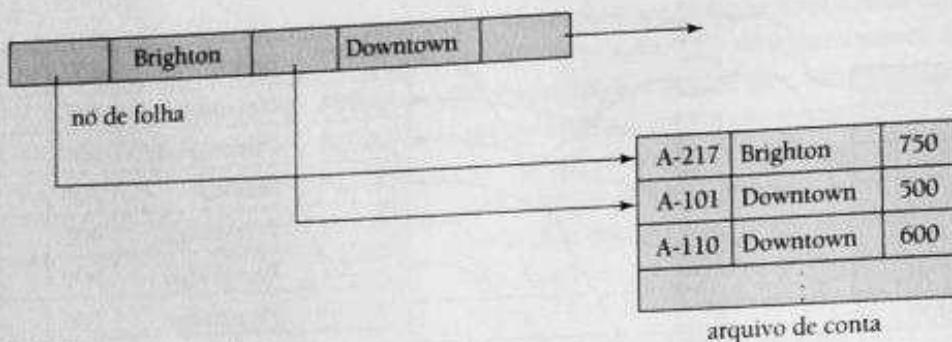


Figura 12.7 Um nó de folha para o índice de árvore B^+ de conta ($n = 3$).

chave de busca que elas contêm, usamos P_n para encadear os nós de folha na ordem da chave de busca. Essa ordenação permite o processamento sequencial eficiente do arquivo.

Os nós não-folha da árvore B^+ formam um índice multi-nível (esparso) sobre os nós de folha. A estrutura dos nós não-folha é igual a dos nós de folha, exceto que todos os ponteiros são ponteiros para nós de folha. Um nó não-folha pode manter até n ponteiros, e precisa manter pelo menos $\lceil n/2 \rceil$ ponteiros. O número de ponteiros em um nó é chamado de *fanout* do nó.

Vamos considerar um nó contendo m ponteiros. Para $i = 2, 3, \dots, m-1$, o ponteiro P_i aponta para a subárvore que contém valores de chave de busca menores que K_i e maiores ou iguais a K_{i-1} . O ponteiro P_m aponta para a parte da subárvore que contém os valores de chave maiores ou iguais a K_{m-1} , e o ponteiro P_1 aponta para a parte da subárvore que contém os valores de chave de busca menores que K_1 .

Ao contrário dos outros nós de folha, o nó raiz pode manter menos de $\lceil n/2 \rceil$ ponteiros; porém, ele precisa manter pelo menos dois ponteiros, a menos que a árvore consista em apenas um nó. Sempre é possível construir uma árvore B^+ , para qualquer n , que satisfaça os requisitos anteriores. A Figura 12.8 mostra uma árvore B^+ completa para o arquivo de conta ($n = 3$). Para simplificar, omitimos os dois ponteiros do próprio arquivo e os ponteiros nulos.

nos de $\lceil n/2 \rceil$ valores, a Figura 12.9 mostra uma árvore B^+ para o arquivo de conta com $n = 5$.

Esses exemplos de árvores B^+ estão bem平衡ados. Ou seja, o tamanho de cada caminho da raiz para o nó de folha é igual. Essa propriedade é um requisito para uma árvore B^+ . Na realidade, o “B” de árvore B^+ significa “balanceada”. É a propriedade de balanceamento das árvores B^+ que garante o bom desempenho para pesquisa, inserção e exclusão.

Consultas em árvores B^+

Vamos considerar como processar consultas sobre uma árvore B^+ . Suponha que queremos encontrar todos os registros com um valor de chave de busca V . A Figura 12.10 apresenta o pseudocódigo para fazer isso. Intuitivamente, o procedimento funciona da seguinte maneira. Primeiro, examinamos o nó raiz, procurando o menor valor de chave de busca que seja maior que V . Suponha que descobrimos que esse valor de chave de busca é K_1 . Depois, acompanhamos o ponteiro P_1 para outro nó. Se não encontrarmos esse valor, então $k \geq K_{m-1}$, onde m é o número de ponteiros no nó. Nesse caso, acompanhamos P_m para outro nó. No nó que alcançamos, novamente procuramos o menor valor de chave de busca maior que V , e mais uma vez seguimos o ponteiro correspondente, como antes. Por fim, alcançamos o nó de folha. No nó de folha, se encontrarmos o valor de

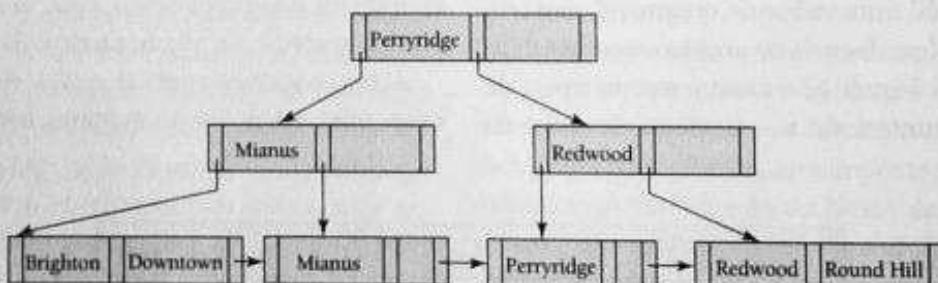


Figura 12.8 Árvore B^+ para arquivo de conta ($n = 3$).

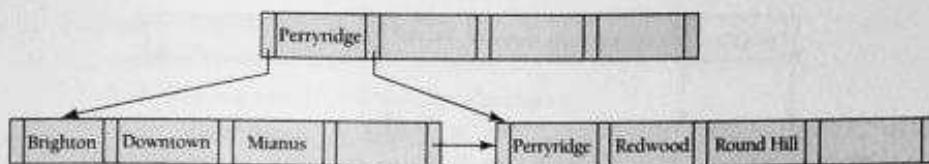


Figura 12.9 Árvore B⁺ para arquivo de conta com $n = 5$.

chave de busca K_i igual a V , então o ponteiro P_i nos levará ao registro ou bucket desejado. Se o valor V não for encontrado no nó de folha, não existirá um registro com o valor de chave V .

Assim, no processamento de uma consulta, atravessamos um caminho na árvore desde a raiz até algum nó de folha. Se houver K valores de chave de busca no arquivo, o caminho não é maior do que $\lceil \log_{n/2}(K) \rceil$.

Na prática, somente alguns poucos nós precisarão ser acessados. Normalmente, um nó tem o mesmo tamanho de um bloco de disco, que normalmente é de 4 kilobytes. Com uma chave de busca com 12 bytes e um ponteiro de disco com 8 bytes, n fica em torno de 200. Mesmo com uma estimativa mais conservadora de 32 bytes para o tamanho da chave de busca, n fica em torno de 100. Com $n = 100$, se tivermos um milhão de valores de chave de busca no arquivo, uma pesquisa exigirá apenas $\lceil \log_{50}(1.000.000) \rceil = 4$ nós a serem acessados. Assim, no máximo quatro blocos precisam ser lidos do disco para a consulta. O nó raiz da árvore normalmente é muito acessado, e provavelmente estará no buffer, de modo que normalmente apenas três ou menos blocos precisam ser lidos do disco.

Uma diferença importante entre estruturas de árvore B⁺ e estruturas de árvore na memória, como árvores binárias, é o tamanho de um nó e, como resultado, a altura da árvore. Em uma árvore binária, cada nó é pequeno e tem no máxi-

mo dois ponteiros. Em uma árvore B⁺, cada nó é grande – normalmente, um bloco de disco – e pode ter uma grande quantidade de ponteiros. Assim, as árvores B⁺ costumam ser baixas e largas, ao contrário das árvores binárias altas e estreitas. Em uma árvore binária balanceada, o caminho para uma pesquisa pode ter tamanho $\lceil \log_2(K) \rceil$, onde K é o número de valores de chave de busca. Com $K = 1.000.000$, como no exemplo anterior, uma árvore binária balanceada exige em torno de 20 acessos de nó. Se cada nó estivesse em um bloco de disco diferente, 20 leituras de bloco seriam necessárias para processar uma pesquisa, ao contrário das quatro leituras de bloco para a árvore B⁺.

Atualizações em árvores B⁺

A inserções e a exclusão são mais complicadas do que a pesquisa, pois pode ser necessário dividir um nó que se torna muito grande como resultado de uma inserção, ou unir nós (ou seja, combinar nós) se um nó se tornar muito pequeno (menos de $\lceil n/2 \rceil$ ponteiros). Além do mais, quando um nó é dividido ou um par de nós é combinado, temos de garantir que o balanço seja preservado. Para introduzir a idéia por trás da inserção e exclusão em uma árvore B⁺, vamos considerar temporariamente que os nós nunca se tornam muito grandes ou muito pequenos. Sob essa suposição, a inserção e a exclusão são realizadas conforme definido a seguir.

```

procedure find(valor V)
    set C = nó raiz
    while C não é o nó de folha begin
        Let  $K_i$  = menor valor de chave de busca > V, se houver
        if não existe um valor then begin
            Let m = número de ponteiros no nó
            set C = nó apontado por  $P_m$ 
        end
        else set C = o nó apontado por  $P_i$ 
    end
    if existe um valor de chave  $K_i$  em C tal que  $K_i = V$ 
        then ponteiro  $P_i$  nos leva ao registro ou bucket desejado
        else não existe registro com valor de chave k

```

Figura 12.10 Consultando uma árvore B⁺.

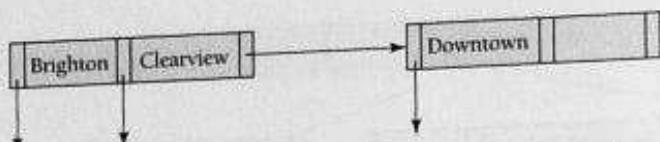


Figura 12.11 Divisão do nó de folha na inserção de "Clearview".

- **Inserção.** Usando a mesma técnica da pesquisa, encontramos o nó de folha em que o valor da chave de busca apareceria. Se o valor da chave de busca já aparecer no nó de folha, acrescentamos o novo registro ao arquivo e, se for preciso, acrescentamos ao bucket um ponteiro para o registro. Se o valor da chave de busca não aparecer, inserimos o valor no nó de folha e o posicionamos de modo que as chaves de busca ainda estejam na ordem. Depois, inserimos o novo registro no arquivo e, se for preciso, criamos um novo bucket com o ponteiro apropriado.
- **Exclusão.** Usando a mesma técnica da pesquisa, encontramos o registro a ser excluído e o removemos do arquivo. Removemos o valor da chave de busca do nó de folha se não houver um bucket associado a esse valor de chave de busca ou se o bucket ficar vazio como resultado da exclusão.

Agora, vamos considerar um exemplo em que um nó precisa ser dividido. Considere que queremos inserir um registro, com um valor de *nome_agência* igual a "Clearview", na árvore B* da Figura 12.8. Usando o algoritmo para pesquisa, descobrimos que "Clearview" deverá aparecer no nó contendo "Brighton" e "Downtown". Portanto, o nó é dividido em dois nós. A Figura 12.11 mostra os dois nós de folha que resultam da inserção de "Clearview" e a divisão do nó contendo "Brighton" e "Downtown". Em geral, apanhamos os n valores de chave de busca (os $n - 1$ valores no nó de folha mais o valor sendo inserido) e colocamos os primeiros $\lceil n/2 \rceil$ no nó existente e os valores restantes em um novo nó.

Tendo dividido um nó de folha, temos de inserir o novo nó de folha na estrutura de árvore B*. Em nosso exemplo, o novo nó tem "Downtown" como seu menor valor de chave

de busca. Temos de inserir esse valor de chave de busca no pai do nó de folha que foi dividido. A árvore B* da Figura 12.12 mostra o resultado da inserção. O valor de chave de busca "Downtown" foi inserido no pai. Foi possível realizar essa inserção porque havia espaço para um valor de chave de busca adicional. Se não houvesse espaço, o pai teria de ser dividido. No pior caso, todos os nós ao longo do caminho até a raiz precisarão ser divididos. Se a própria raiz for dividida, a árvore inteira se torna mais profunda.

A técnica geral para inserção em uma árvore B* é determinar o nó de folha f em que a inserção precisa ocorrer. Se houver uma divisão, insira o novo nó no pai do nó f . Se essa inserção causar uma divisão, suba recursivamente pela árvore até que uma inserção não cause uma divisão ou uma nova raiz seja criada.

A Figura 12.13 esboça o algoritmo de inserção em pseudocódigo. O procedimento *insert* insere um par de ponteiros de valor de chave no índice, usando dois procedimentos subsidiários, *insert_in_leaf* e *insert_in_parent*. No pseudocódigo, K , N , P e T indicam os ponteiros para os nós, com L sendo usado para indicar um nó de folha. $L.K_i$ e $L.P_i$ indicam o i -ésimo valor e o i -ésimo ponteiro no nó L , respectivamente; $T.K_i$ e $T.P_i$ são usados de modo semelhante. O pseudocódigo também utiliza a função *parent(N)* para encontrar o pai de um nó N . Podemos calcular uma lista de nós no caminho a partir da raiz até a folha enquanto encontramos inicialmente o nó de folha, e podemos usá-lo mais tarde para encontrar, de modo eficiente, o pai de qualquer nó no caminho.

O procedimento *insert_in_parent* recebe como parâmetros N , K' , N' , onde o nó N foi dividido em N e N' , com K' sendo o valor de folha em N' . O procedimento modifica o pai de N para registrar a divisão. Os procedimentos *insert_into_index* e *insert_in_parent* utilizam uma área tem-

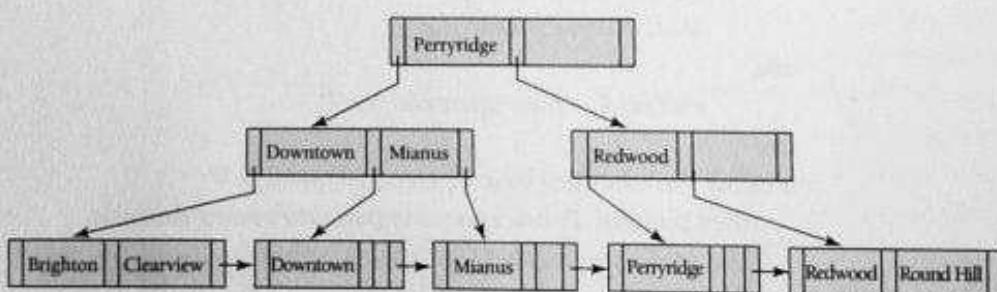


Figura 12.12 Inserção de "Clearview" na árvore B+ da Figura 12.8.

```

procedure insert(valor K, ponteiro P)
    encontre o nó de folha L que deve conter o valor de chave K
    if (L tem menos de n-1 valores de chave)
        then insert_in_leaf (L, K, P)
    else begin /* L tem n-1 valores de chave, divide */
        Crie novo L'
        Copie L.P1 ... L.Kn-1 para um bloco de memória T que
            pode manter n pares (ponteiro, valor de chave)
        insert_in_leaf (T, K, P)
        Defina L'.Pn = L.Pn; Defina L'.Pn = L'
        Apague L.P1 a L.Kn-1 de L
        Copie T.P1 a T.K[n/2] de T para L começando em L.P1
        Copie T.P[n/2]+1 a T.Kn de T para L' começando em L'.P1
        Deixe K' ser o menor valor de chave em L'
        insert_in_parent(L, K', L')
    end

procedure insert_in_leaf (nó L, valor K, ponteiro P)
    if K é menor que L.K1
        then insira P, K em L logo antes de L.P1
    else begin
        Deixe K' ser o menor valor em L que é menor que K
        insira P, K em L logo após T.Ki
    end

procedure insert_in_parent(nó N, valor K', nó N')
    if N é a raiz da árvore
        then begin
            crie novo nó R contendo N, K', N' /* N e N' são ponteiros */
            torne R a raiz da árvore
            return
        end
    Deixe P = parent(N)
    if (P tem menos de N ponteiros)
        then insira (K', N') em P logo após N
    else begin /* Divide P */
        Copie P para um bloco de memória T que pode manter P e (K', N')
        Insira (K', N') em T logo após N
        Apague todas as entradas de P; Crie nó P'
        Copie T.P1...T.P[n/2] para P
        Deixe K'' = T.K[n/2]
        Copie T.P[n/2]+1...T.Pn+1 para P'
        insert_in_parent(P, K'', P')
    end

```

Figura 12.13 Inserção de entrada em uma árvore B+.

poraria da memória T para armazenar o conteúdo de um nó sendo dividido. Os procedimentos podem ser modificados para copiar diretamente os dados do nó sendo dividido para o nó recém-criado, reduzindo o tempo necessário para

copiar os dados. Porém, o uso do espaço temporário T simplifica os procedimentos.

Agora, consideraremos as exclusões que fazem com que os nós de árvore contenham poucos ponteiros. Primeiro, va-

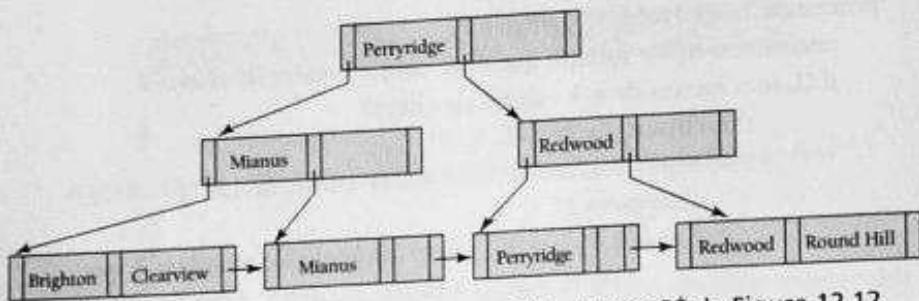


Figura 12.14 Exclusão de “Downtown” da árvore B⁺ da Figura 12.12.

mos excluir “Downtown” da árvore B⁺⁺ da Figura 12.12. Localizamos a entrada para “Downtown” usando nosso algoritmo de pesquisa. Quando excluímos a entrada para “Downtown” do seu nó de folha, a folha fica vazia. Como, em nosso exemplo, $n = 3$ e $0 < \lceil (n-1)/2 \rceil$, esse código precisa ser eliminado da árvore B⁺. Para excluir um nó de folha, temos de excluir o ponteiro para ele a partir do seu pai. Em nosso exemplo, essa exclusão deixa o nó pai, que anteriormente tinha três ponteiros, com apenas dois ponteiros. Como $2 \geq \lceil n/2 \rceil$, o nó ainda é suficientemente grande, e a operação de exclusão está concluída. A árvore B⁺ resultante aparece na Figura 12.14.

Quando fazemos uma exclusão a partir de um pai de um nó de folha, o próprio nó pai pode se tornar muito pequeno. Isso é exatamente o que acontece se excluirmos “Perryridge” da árvore B⁺ da Figura 12.14. A exclusão da entrada Perryridge faz com que um nó de folha fique vazio. Quando excluímos o ponteiro para esse nó no pai desse, o pai fica com apenas um ponteiro. Como $n = 3$, $\lceil n/2 \rceil = 2$, e com isso apenas um ponteiro é muito pouco. Porém, como o nó pai contém informações úteis, não podemos simplesmente exclui-lo. Em vez disso, examinamos o nó irmão (o nó não-folha contendo a única chave de busca, Mianus). Esse nó irmão tem espaço para acomodar a informação contida em nosso nó agora pequeno, então unimos esses nós de modo que o nó irmão agora contenha as chaves “Mianus” e “Redwood”. O outro nó (aquele contendo apenas a chave de busca “Redwood”) agora contém informações redundantes e pode ser excluído do seu pai (que, em nosso exemplo, é a raiz). A Figura 12.15 mostra o resultado. Observe que a raiz tem apenas um ponteiro filho após a exclusão, de modo que é excluída e seu único

filho se torna a raiz. Assim, a profundidade da árvore B⁺ foi diminuída em 1.

Nem sempre é possível unir nós. Como ilustração, exclua “Perryridge” da árvore B⁺ da Figura 12.12. Neste exemplo, a entrada “Downtown” ainda faz parte da árvore. Mais uma vez, o nó de folha contendo “Perryridge” torna-se vazio. O pai do nó de folha se torna muito pequeno (apenas um ponteiro). Porém, neste exemplo, o nó irmão já contém o número máximo de ponteiros: três. Assim, ele não pode acomodar um ponteiro adicional. A solução nesse caso é redistribuir os ponteiros de modo que cada irmão tenha dois ponteiros. O resultado aparece na Figura 12.16. Observe que a redistribuição de valores necessita de uma mudança de um valor de chave de busca no pai dos dois irmãos.

Em geral, para excluir um valor em uma árvore B⁺, realizamos uma pesquisa no valor e o excluímos. Se o nó for muito pequeno, o excluímos do seu pai. Essa exclusão resulta na aplicação recursiva do algoritmo de exclusão até que a raiz seja alcançada, um pai permanece adequadamente cheio após a exclusão, ou após a redistribuição ser aplicada.

A Figura 12.17 esboça o pseudocódigo para a exclusão a partir de uma árvore B⁺. O procedimento `swap_variables(N,N')` simplesmente troca os valores das variáveis (de ponteiro) N e N'; essa troca não tem efeito sobre a própria árvore. O pseudocódigo usa a condição “muito poucos ponteiros/valores”. Para nós não de folha, esse critério significa menos de $\lceil n/2 \rceil$ ponteiros; para nós de folha, isso significa menos de $\lceil (n-1)/2 \rceil$ valores. O pseudocódigo redistribui as entradas pedindo emprestada uma única entrada de um nó adjacente. Também podemos redistribuir as entra-

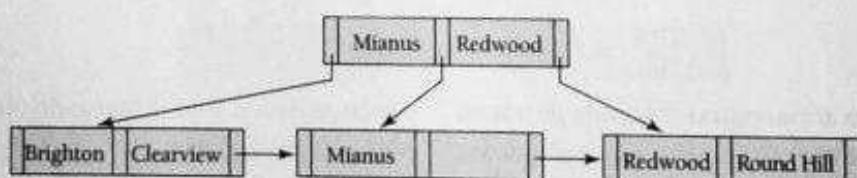


Figura 12.15 Exclusão de “Perryridge” da árvore B⁺ da Figura 12.14.

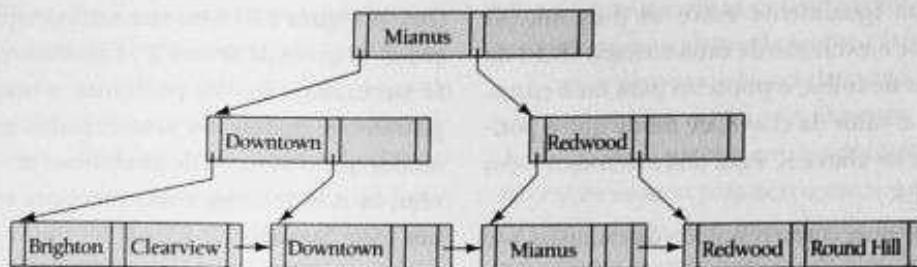


Figura 12.16 Exclusão de "Perryridge" da árvore B+ da Figura 12.12.

```

procedure delete(valor K, ponteiro P)
    encontre o nó de folha L que contém (K,P)
    delete_entry(L, K, P)

procedure delete_entry(nó N, valor K, ponteiro P)
    exclua (K,P) de N
    if (N é a raiz and N tem apenas um filho restante)
    then torna o filho de N a nova raiz da árvore e exclui N
    else if (N tem poucos valores/ponteiros) then begin
        deixe N' ser o filho anterior ou seguinte de parent(N)
        deixe K' ser o valor entre os ponteiros N e N' em parent(N)
        if (entradas em N e N' podem caber em um único nó)
            then begin /* Une nós */
                if (N é predecessor de N') then swap_variables(N, N')
                if (N não é uma folha)
                    then anexa K' e todos os ponteiros e valores em N a N'
                    else anexa todos os pares (Ki,Pi) em N a N'; define N'.Pn = N.Pn
                delete_entry(parent(N), K', N); exclui nó N
            end
        else begin /* Redistribuição: pega emprestada entrada de N' */
            if (N' é predecessor de N) then begin
                if (N é um nó não de folha) then begin
                    deixe m ser tal que N'.Pm seja o último ponteiro em N'
                    remova (N'.Km-1, N'.Pm) de N'
                    insira (N'.Pm, K') como primeiro ponteiro e valor em N,
                        deslocando outros ponteiros e valores para a direita
                    substitua K' em parent(N) por N'.Km-1
                end
            else begin
                deixe m ser tal que (N'.Pm, N'.Km) seja o último par
                    ponteiro/valor em N'
                remova (N'.Pm, N'.Km) de N'
                insira (N'.Pm, N'.Km) como primeiro ponteiro e valor em N,
                    deslocando outros ponteiros e valores para a direita
                substitua K' em parent(N) por N'.Km
            end
        end
    else ... simétrico ao caso then ...
end

```

Figura 12.17 Exclusão da entrada de uma árvore B+.

das reparticionando-as igualmente entre os dois nós. O pseudocódigo refere-se à exclusão de uma entrada (K, P) de um nó. No caso de nós de folha, o ponteiro para uma entrada realmente precede o valor da chave, de modo que o ponteiro P precede o valor de chave K . Para nós internos, P vem após o valor de chave K .

Vale a pena observar que, como resultado da exclusão, um valor de chave que está presente em um nó interno da árvore B^+ pode não estar presente em qualquer folha da árvore.

Embora as operações de inserção e exclusão sobre as árvores B^+ sejam complicadas, elas exigem relativamente poucas operações de E/S, que é um benefício importante, pois as operações de E/S são dispendiosas. Pode-se mostrar que o número de operações de E/S necessárias para a inserção ou exclusão no pior caso é proporcional a $\log_{n/2}(K)$, onde n é o número máximo de ponteiros em um nó, e K é o número de valores da chave de busca. Em outras palavras, o custo das operações de inserção e exclusão é proporcional à altura da árvore B^+ , e, portanto, é baixo. É a velocidade da operação sobre as árvores B^+ que as torna uma estrutura de índice frequentemente utilizada nas implementações de banco de dados.

Organização de arquivos de árvore B^+

Conforme mencionamos na seção “Arquivos de índice de árvore B^+ ”, a principal desvantagem da organização de arquivo sequencial indexada é a degradação do desempenho quando o arquivo cresce: com o crescimento, uma porcentagem maior de registros de índice e registros reais se torna fora de ordem, e é armazenada em blocos de estouro. Solucionamos a degradação das pesquisas de índice usando índices de árvore B^+ no arquivo. Solucionamos o problema da degradação para armazenar os registros reais usando o nível de folha da árvore B^+ a fim de organizar os blocos que contêm os registros reais. Usamos a estrutura de árvore B^+ não apenas como um índice, mas também como um organizador para registros em um arquivo. Em uma organização de arquivo de árvore B^+ , os nós de folha da árvore armazenam registros, em vez de armazenar ponteiros para regis-

tos. A Figura 12.18 mostra um exemplo de uma organização de arquivo de árvore B^+ . Como os registros normalmente são maiores que os ponteiros, o número máximo de registros que podem ser armazenados em um nó de folha é menor que o número de ponteiros em um nó não-folha. Portanto, os nós de folha ainda precisam estar cheios pelo menos até a metade.

A inserção e a exclusão de registros de uma organização de arquivo de árvore B^+ são tratadas da mesma maneira que a inserção e a exclusão de entradas em um índice de árvore B^+ . Quando um registro com determinado valor de chave v é inserido, o sistema localiza o bloco que deverá conter o registro procurando na árvore B^+ a maior chave na árvore que é $\leq v$. Se o bloco localizado tiver espaço livre suficiente para o registro, o sistema armazena o registro no bloco. Caso contrário, como na inserção da árvore B^+ , o sistema divide o bloco em dois e redistribui os registros nele (na ordem da chave da árvore B^+) para criar espaço para o novo registro. A divisão se propaga para cima na árvore B^+ , pelo modo normal. Quando excluímos um registro, o sistema primeiro o remove do bloco que o contém. Se um bloco B ficar com menos de metade da ocupação como resultado, os registros em B são redistribuídos com os registros em um bloco B' adjacente. Considerando registros de tamanho fixo, cada bloco terá pelo menos metade da quantidade máxima de registros que pode admitir. O sistema atualiza os nós não de folha da árvore B^+ pelo modo normal.

Quando usamos uma árvore B^+ para a organização de arquivo, a utilização do espaço é particularmente importante, pois o espaço ocupado pelos registros provavelmente é muito mais do que o espaço ocupado pelas chaves e ponteiros. Podemos melhorar a utilização do espaço em uma árvore B^+ envolvendo mais nós irmãos na redistribuição durante as divisões e mesclagens. A técnica se aplica aos nós de folha e aos nós internos, e funciona da seguinte maneira.

Durante a inserção, se um nó estiver cheio, o sistema tenta redistribuir algumas de suas entradas para um dos nós adjacentes, a fim de criar espaço para uma nova entrada. Se essa tentativa falhar porque os próprios nós adjacentes estão cheios, o sistema divide o nó e divide as entradas

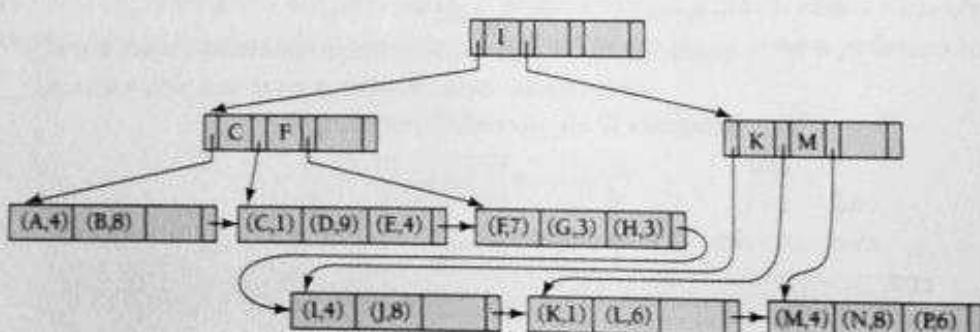


Figura 12.18 Organização de arquivo de árvore B^+ .

por igual entre um dos nós adjacentes e os dois nós que obteve dividindo o nó original. Como os três nós juntos contêm mais um registro do que pode caber nos dois nós, cada nó estará cerca de dois terços cheio. Mais precisamente, cada nó terá pelo menos $\lfloor 2n/3 \rfloor$ entradas, onde n é o número máximo de entradas que o nó pode manter. ($\lfloor x \rfloor$ indica o maior inteiro que é menor ou igual a x , ou seja, removemos a parte fracionária, se houver.)

Durante a exclusão de um registro, se a ocupação de um nó ficar abaixo de $\lfloor 2n/3 \rfloor$, o sistema tenta pegar emprestada uma entrada de um dos nós irmãos. Se os dois nós irmãos tiverem $\lfloor 2n/3 \rfloor$ registros, em vez de pegar emprestada uma entrada, o sistema redistribui as entradas no nó e nos dois irmãos por igual entre dois dos nós, e exclui o terceiro nó. Podemos usar essa técnica porque o número total de entradas é $3\lfloor 2n/3 \rfloor - 1$, que é menor que $2n$. Com três nós adjacentes sendo usados para a redistribuição, podemos garantir que cada nó terá $\lfloor 3n/4 \rfloor$ entradas. Em geral, se m nós ($(m-1$ irmãos) estiverem envolvidos na redistribuição, pode-se garantir que cada nó terá pelo menos $\lfloor (m-1)n/m \rfloor$ entradas. Entretanto, o custo da atualização se torna maior à medida que mais nos irmãos são envolvidos na redistribuição.

Observe que, em um índice ou organização de arquivo de árvore B*, os nós de folha que são adjacentes entre si na árvore podem estar localizados em diferentes locais do disco. Quando uma organização de arquivo é recém-criada em um conjunto de registros, é possível alocar blocos que são principalmente contíguos no disco aos nós de folha que são contíguos na árvore. Assim, uma varredura sequencial dos nós de folha corresponderia a uma varredura geralmente sequencial no disco. À medida que ocorrem inserções e exclusões na árvore, a sequencialidade é perdida cada vez mais, e o acesso sequencial precisa esperar pelas buscas de disco cada vez mais frequentemente. Para restaurar a sequencialidade, pode ser necessária uma reconstrução de índice.

Organizações de árvore B* podem ser usadas para armazenar grandes objetos, como clobs e blobs SQL, que podem ser maiores do que um bloco de disco, contendo até vários gigabytes. Esses objetos grandes podem ser armazenados dividindo-os em sequências de registros menores que são colocados em uma organização de arquivo de árvore B*. Os registros podem ser numerados sequencialmente, ou numerados pelo deslocamento de bytes do registro dentro do objeto grande, e o número do registro pode ser usado como a chave de busca.

Indexando strings

A criação de índices de árvore B* sobre atributos com valor de string ocasiona dois problemas. O primeiro problema é que as strings podem ter tamanho variável. O segundo pro-

blema é que as strings podem ser longas, levando a um baixo fanout e uma altura de árvore consequentemente maior.

Com as chaves de busca de tamanho variável, diferentes nós podem ter diferentes fanouts, mesmo que estejam cheios. Um nó precisa ser dividido se estiver cheio, ou seja, não existe espaço para acrescentar uma nova entrada, independente de quantas entradas de busca ele tenha. De modo semelhante, os nós podem ser mesclados ou as entradas redistribuídas, dependendo da fração de espaço utilizada nos nós, em vez de nos basearmos no número máximo de entradas que o nó pode aceitar.

O fanout dos nós pode ser aumentado usando uma técnica chamada compactação de prefixo. Com a compactação de prefixo, não armazenamos o valor inteiro da chave de busca nos nós internos. Só armazenamos um prefixo de cada valor de chave de busca que seja suficiente para distinguir entre os valores de chave nas subárvores que ela separa. Por exemplo, se tivéssemos um índice sobre nomes, o valor de chave em um nó interno poderia ser um prefixo de um nome; pode ser suficiente armazenar "Silb" em um nó interno, em vez do nome "Silberschatz" completo, se os valores mais próximo nas duas subárvores que ela separa forem, digamos, "Silas" e "Silver", respectivamente.

Arquivos de índice de árvore B

Índices de árvore B são semelhantes aos índices de árvore B*. A principal distinção entre as duas técnicas é que uma árvore B elimina o armazenamento redundante de valores de chave de busca. Na árvore B* da Figura 12.12, as chaves de busca "Downtown", "Mianus", "Redwood" e "Perryridge" aparecem duas vezes. Cada valor de chave de busca aparece em algum nó de folha; várias são repetidas nos nós não-folha.

Uma árvore B permite que os valores de chave de busca apareçam apenas uma vez. A Figura 12.19 mostra uma árvore B que representa as mesmas chaves de busca da árvore B* da Figura 12.12. Como as chaves de busca não são repetidas na árvore B, podemos armazenar o índice em menos nós de árvore do que no índice de árvore B* correspondente. Porém, como as chaves de busca que aparecem em nós não-folha não aparecem em outro lugar na árvore B, somos forçados a incluir um campo ponteiro adicional para cada chave de busca em um nó não-folha. Esses ponteiros adicionais apontam para os registros de arquivo ou buckets para a chave de busca associada.

Um nó de folha generalizado da árvore B aparece na Figura 12.20a; um nó não-folha aparece na Figura 12.20b. Os nós de folha são iguais aos das árvores B*. Nos nós não-folha, os ponteiros P_j são os ponteiros de árvore que usamos também para árvores B*, enquanto os ponteiros B_i são ponteiros de bucket ou registro de arquivo. Na árvore B generalizada da figura, existem $n-1$ chaves no nó de folha,

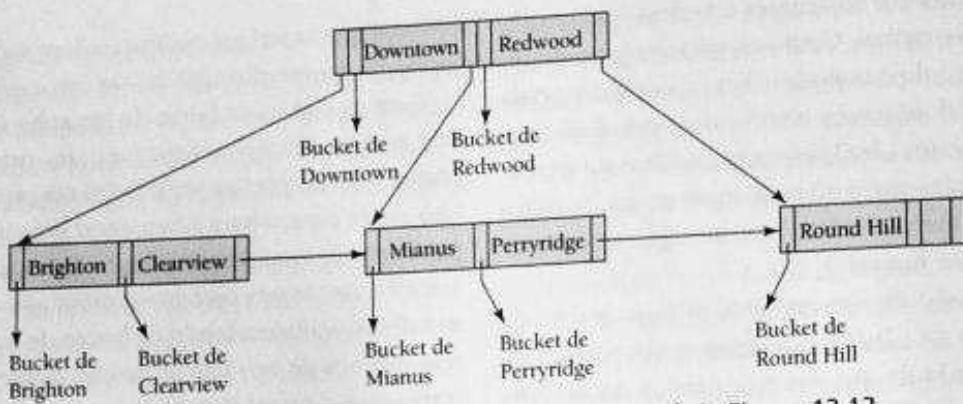


Figura 12.19 Árvore B equivalente da árvore B⁺ da Figura 12.12.

mas existem $m-1$ chaves no nó não-folha. Essa discrepância ocorre porque os nós não-folha precisam incluir ponteiros B_i , reduzindo assim o número de chaves de busca que podem ser mantidas nesses nós. Claramente, $m < n$, mas o relacionamento exato entre m e n depende do tamanho relativo das chaves de busca e ponteiros.

O número de nós acessados em uma pesquisa em uma árvore B depende de onde a chave de busca está localizada. Uma pesquisa sobre uma árvore B⁺ exige a travessia de um caminho da raiz da árvore até algum nó de folha. Ao contrário, às vezes é possível encontrar o valor desejado em uma árvore B antes de alcançar um nó de folha. Porém, aproximadamente n vezes mais chaves são armazenadas no nível de folha de uma árvore B do que nos níveis não-folha e, como n normalmente é grande, o benefício de localizar certos valores mais cedo é relativamente pequeno. Além do mais, o fato de que menos chaves de busca aparecem em um nó de árvore B não-folha, em comparação com as árvores B⁺, significa que uma árvore B tem um fanout menor e, portanto, pode ter profundidade maior do que a da árvore B⁺ correspondente. Assim, a pesquisa em uma árvore B é mais rápida para algumas chaves de busca, porém mais lenta para outras, embora, em geral, o tempo de pesquisa ainda seja proporcional ao logaritmo do número de chaves de busca.

A exclusão em uma árvore B é mais complicada. Em uma árvore B⁺, a entrada excluída sempre aparece em uma folha. Em uma árvore B, a entrada excluída pode aparecer em um

nó não-folha. O valor correto precisa ser selecionado como um substituto a partir da subárvore do nó contendo a entrada excluída. Especificamente, se a chave de busca K_i for excluída, a menor chave de busca que aparece na subárvore do ponteiro P_{i+1} precisa ser movida para o campo anteriormente ocupado por K_i . Outras ações precisam ser tomadas se o nó de folha agora tiver muito poucas entradas. Ao contrário, a inserção em uma árvore B só é ligeiramente mais complicada do que a inserção em uma árvore B⁺.

As vantagens de espaço das árvores B são poucas para índices grandes, e normalmente não superam as desvantagens que observamos. Assim, muitos implementadores de sistema de banco de dados preferem a simplicidade estrutural de uma árvore B⁺. Os exercícios exploram os detalhes dos algoritmos de inserção e exclusão para árvores B.

Acesso por chave múltipla

Até agora, consideramos implicitamente que apenas um índice no atributo é usado para processar uma consulta em uma relação. Porém, para certos tipos de consultas, é vantajoso usar vários índices se eles existem, ou usar um índice baseado em uma chave de busca de múltiplos atributos.

Usando múltiplos índices de chave única

Suponha que o arquivo de conta tenha dois índices: um para nome_agência e um para saldo. Considere a seguinte

P_1	K_1	P_2	...	P_{n-1}	K_{n-1}	P_n
-------	-------	-------	-----	-----------	-----------	-------

(a)

P_1	B_1	K_1	P_2	B_2	K_2	...	P_{m-1}	B_{m-1}	K_{m-1}	P_m
-------	-------	-------	-------	-------	-------	-----	-----------	-----------	-----------	-------

(b)

Figura 12.20 Nós típicos de uma árvore B. (a) Nô de folha. (b) Nô não de folha.

consulta: "Encontrar todos os números de conta na agência Perryridge com saldos iguais a \$1.000". Escrevemos

```
select número_empréstimo
from conta
where nome_agência = "Perryridge" and saldo = 1000
```

Existem três estratégias possíveis para processar essa consulta:

1. Usar o índice sobre *nome_agência* e encontrar todos os registros pertencentes à agência Perryridge. Examinar cada um desses registros para ver se *saldo* = 1000.
2. Usar o índice sobre *saldo* para encontrar todos os registros pertencentes às contas com saldos iguais a \$1.000. Examinar cada um desses registros para ver se *nome_agência* = "Perryridge".
3. Usar o índice sobre *nome_agência* para encontrar ponteiros para todos os registros pertencentes à agência Perryridge. Além disso, usar o índice sobre *saldo* para encontrar ponteiros para todos os registros pertencentes às contas com um saldo igual a \$1.000. Apanhar a interseção desses dois conjuntos de ponteiros. Aqueles ponteiros que estão na interseção apontam para os registros pertencentes à Perryridge e contas com um saldo de \$1.000.

A terceira estratégia é a única das três que tira proveito da existência de vários índices. Porém, até mesmo essa estratégia pode ser uma escolha fraca se acontecer o seguinte:

- Existem muitos registros pertencentes à agência Perryridge.
- Existem muitos registros pertencentes a contas com um saldo de \$1.000.
- Existem apenas alguns registros pertencentes tanto à agência Perryridge quanto a contas com um saldo de \$1.000.

Se essas condições forem mantidas, temos de varrer uma grande quantidade de ponteiros para produzir um resultado pequeno. Uma estrutura de índice chamada "índice de mapa de bits" pode, em alguns casos, agilizar bastante a operação de interseção usada na terceira estratégia. Os índices de mapa de bits são esboçados na seção "Índices de mapa de bits".

Índices sobre chaves múltiplas

Uma estratégia alternativa para esse caso é criar e usar um índice sobre uma chave de busca (*nome_agência*, *saldo*) –

ou seja, a chave de busca consistindo no nome da agência concatenado com o saldo da conta. Tal chave de busca, contendo mais de um atributo, às vezes é chamada de **chave de busca composta**. A estrutura do índice é igual à de qualquer outro índice, mas a única diferença é que a chave de busca não é um único atributo, mas sim uma lista de atributos. A chave de busca pode ser representada como uma tupla de valores, da forma (a_1, \dots, a_n) , onde os atributos indexados são A_1, \dots, A_n . A ordenação dos valores da chave de busca é a ordenação lexicográfica. Por exemplo, para o caso de chaves de busca com dois atributos, $(a_1, a_2) < (b_1, b_2)$ se $a_1 < b_1$ ou $a_1 = b_1$ e $a_2 < b_2$. A ordenação lexicográfica é basicamente o mesmo que ordenação alfabética de palavras.

Podemos usar um índice ordenado (árvore B⁺) para responder de forma eficiente a consultas da forma

```
select número_empréstimo
from conta
where nome_agência = 'Perryridge'
and saldo = 1000
```

Consultas como a seguinte, que especifica uma condição de igualdade sobre o primeiro atributo da chave de busca (*nome_agência*) e um intervalo sobre o segundo atributo da chave de busca (*saldo*), também podem ser tratadas de forma eficiente, pois correspondem a uma consulta de intervalo sobre o atributo de busca.

```
select número_empréstimo
from conta
where nome_agência = 'Perryridge'
and saldo < 1000
```

Podemos até mesmo usar um índice ordenado sobre a chave de busca (*nome_agência*, *saldo*) para responder a seguinte consulta sobre apenas um atributo de forma eficiente:

```
select número_empréstimo
from conta
where nome_agência = 'Perryridge'
```

Uma condição de igualdade *nome_agência* = "Perryridge" é equivalente a uma consulta de intervalo sobre o intervalo com a extremidade inferior (Perryridge, $-\infty$) e a extremidade superior (Perryridge, $+\infty$). As consultas com intervalo sobre apenas o atributo *nome_agência* podem ser tratadas de maneira semelhante.

Contudo, o uso de uma estrutura de índice ordenado sobre uma chave de busca composta tem algumas limitações. Como ilustração, considere a consulta

```
select número_empréstimo
from conta
where nome_agência < "Perryridge"
and saldo = 1000
```

Podemos responder a essa pergunta usando um índice ordenado sobre a chave de busca (*nome_agência, balance*): para cada valor de *nome_agência* que seja menor que "Perryridge" em ordem alfabética, o sistema localiza registros com um valor de *saldo* igual a 1.000. Porém, cada registro provavelmente estará em um bloco de disco diferente, devido à ordenação dos registros no arquivo, levando a muitas operações de E/S.

A diferença entre essa consulta e as duas anteriores é que a condição no primeiro atributo (*nome_agência*) é uma condição de comparação, em vez de uma condição de igualdade. A condição não corresponde a uma consulta de intervalo sobre a chave de busca.

Para agilizar o processamento de consultas gerais de chave de busca composta (que podem envolver uma ou mais operações de comparação), podemos usar várias estruturas especiais. Vamos considerar os *índices de mapa de bits* na seção "Índices de mapa de bits". Existe outra estrutura, chamada *árvore R*, que pode ser usada para essa finalidade. A árvore R é uma extensão da árvore B⁺ para lidar com a indexação sobre várias dimensões. Como a árvore R é usada principalmente com tipos de dados geográficos, descrevemos a estrutura no Capítulo 24.

Chaves de busca não exclusivas

A criação de buckets de ponteiros de registro para lidar com chaves de busca nãoexclusivas (ou seja, chaves de busca que podem lidar com mais de um registro combinando) cria várias complicações quando as árvores B⁺ são implementadas. Se os buckets forem mantidos no nó de folha, um código extra é necessário para lidar com os buckets de tamanho variável e para lidar com buckets que se tornam maiores do que o tamanho do nó de folha. Se os buckets forem armazenados em páginas separadas, uma operação de E/S extra pode ser necessária para apanhar os registros.

Uma solução simples para esse problema, usada pela maioria dos sistemas de banco de dados, é tornar as chaves de busca exclusivas, acrescentando um atributo exclusivo extra à chave de busca. O valor do atributo extra poderia ser uma id de registro (se o sistema de banco de dados admitir ids de registro), ou apenas um número que seja exclusivo entre todos os registros com o mesmo valor de chave de busca. Por exemplo, se tivéssemos um índice sobre o atributo *nome_cliente* da tabela *depositante*, as entradas correspondentes a um nome de cliente em particular teriam

diferentes valores para o atributo extra. Com isso, a chave de busca estendida terá garantias de exclusividade.

Uma busca com o atributo de chave de busca original se torna uma pesquisa de intervalo sobre a chave de busca estendida, como vimos na seção anterior, o valor do atributo extra é ignorado durante a pesquisa.

Índices de cobertura

Índices de cobertura são índices que armazenam os valores de alguns atributos (fora os atributos da chave de busca) junto com os ponteiros do registro. Armazenar valores de atributo extras é útil com os índices secundários, pois nos permitem responder a algumas perguntas usando apenas o índice, sem sequer pesquisar os registros reais.

Por exemplo, suponha que tenhamos um índice não agrupado sobre o atributo *número_conta* da relação *conta*. Se armazenarmos o valor do atributo *saldo* junto com o ponteiro do registro, podemos responder a consultas que exijam o *saldo* (mas não o outro atributo, *nome_agência*) sem acessar o registro *conta*.

O mesmo efeito poderia ser obtido criando-se um índice sobre a chave de busca (*número_conta, saldo*), mas um índice de cobertura reduz o tamanho da chave de busca, permitindo um fanout maior nos nós internos, e potencialmente reduzindo a altura do índice.

Índices secundários e relocação de registros

Algumas organizações de arquivo, como a organização de arquivo de árvore B⁺, podem mudar o local dos registros mesmo quando os registros não foram atualizados. Como um exemplo, quando uma página de folha é dividida em uma organização de arquivo de árvore B⁺, uma série de registros são movidos para uma nova página. Nesses casos, todos os índices secundários que armazenam ponteiros para os registros relocados teriam de ser atualizados, embora os valores nos registros possam não ter alterado. Cada página de folha pode conter uma quantidade muito grande de registros, e cada um deles pode ser em locais diferentes em cada índice secundário. Assim, a divisão de página de folha pode exigir dezenas ou mesmo centenas de operações de E/S para atualizar todos os índices secundários afetados, tornando-a uma operação muito dispendiosa.

Uma técnica para lidar com esse problema é a seguinte. Nos índices secundários, no lugar de ponteiros para os registros indexados, armazenamos os valores dos atributos de chave de busca do índice primário. Por exemplo, suponha que tenhamos um índice primário sobre o atributo *número_conta* da relação *conta*; então, um índice secundário sobre *nome_agência* armazenaria com cada nome de agência uma

lista de valores de *numero_conta* dos registros correspondentes, em vez de armazenar ponteiros para os registros.

A relocação de registros devido a divisões de página de folha não exige qualquer atualização sobre tal índice secundário. Porém, localizar um registro usando o índice secundário agora exige duas etapas: primeiro, usamos o índice secundário para encontrar os valores de chave de busca do índice primário, e depois usamos o índice primário para encontrar os registros correspondentes.

Essa técnica, portanto, reduz bastante o custo da atualização de índice, devido à reorganização do arquivo, embora aumente o custo do acesso aos dados, usando um índice secundário.

Hashing estático

Uma desvantagem da organização de arquivo sequencial é que temos de acessar uma estrutura de índice para localizar dados, ou temos de usar a busca binária, e isso resulta em mais operações de E/S. As organizações de arquivo baseadas na técnica de hashing permitem evitar o acesso a uma estrutura de índice. O hashing também oferece um modo de construir índices. Estudaremos as organizações de arquivo e os índices baseados em hashing nas próximas seções.

Em nossa descrição do hashing, usaremos o termo bucket para indicar uma unidade de armazenamento que pode armazenar um ou mais registros. Um bucket normalmente é um bloco de disco, mas poderia ser menor ou maior do que um bloco de disco.

Formalmente, considere que K indica o conjunto de todos os valores de chave de busca, e que B indica o conjunto de todos os endereços de bucket. Uma função de hash h é uma função de K para B . Considere que h indica uma função de hash.

Para inserir um registro com chave de busca K_i , calculamos $h(K_i)$, que oferece o endereço do bucket para esse registro. Suponha, por enquanto, que existe espaço no bucket para armazenar o registro. Então, o registro é armazenado nesse bucket.

Para realizar uma pesquisa sobre o valor de chave de busca K_i , simplesmente calculamos $h(K_i)$, depois pesquisamos o bucket com esse endereço. Suponha que duas chaves de busca, K_5 e K_7 , tenham o mesmo valor de hash; ou seja, $h(K_5) = h(K_7)$. Se realizarmos uma pesquisa sobre K_5 , o bucket $h(K_5)$ contém registros com valores de chave de busca K_5 e registros com valores de chave de busca K_7 . Assim, temos de verificar o valor de chave de busca de cada registro no bucket para verificar se o registro é aquele que desejamos.

A exclusão é igualmente simples. Se o valor de chave de busca do registro a ser excluído for K_i , calculamos $h(K_i)$, depois consultamos o bucket correspondente para esse registro e excluímos o registro do bucket.

O hashing pode ser usado para duas finalidades diferentes. Em uma organização de arquivo de hash, obtemos o endereço do bloco de disco contendo um registro desejado diretamente calculando uma função sobre o valor de chave de busca do registro. Em uma organização de índice de hash, organizamos as chaves de busca, com seus ponteiros associados, para uma estrutura de arquivo de hash.

Funções de hash

A pior função de hash possível mapeia todos os valores de chave de busca para o mesmo bucket. Essa função é indesejável porque todos os registros precisam ser mantidos no mesmo bucket. Uma pesquisa precisa examinar cada registro desse tipo para encontrar o desejado. A função de hash ideal distribui as chaves armazenadas uniformemente por todos os buckets, de modo que cada um tenha o mesmo número de registros.

Como não sabemos, durante o projeto, exatamente quais valores de chave de busca serão armazenados no arquivo, queremos escolher uma função de hash que atribua valores de chave de busca aos buckets de modo que a distribuição tenha estas qualidades:

- A distribuição é uniforme. Ou seja, a função de hash atribui a cada bucket o mesmo número de valores de chave de busca do conjunto de todos os valores de chave de busca possíveis.
- A distribuição é aleatória. Ou seja, no caso mais comum, cada bucket terá quase o mesmo número de valores atribuídos a ele, independente da distribuição real dos valores de chave de busca. Mais precisamente, o valor de hash não estará relacionado a qualquer ordenação externamente visível sobre os valores da chave de busca, como a ordenação alfabética ou a ordenação pelo tamanho das chaves de busca; a função de hash parecerá ser aleatória.

Como ilustração desses princípios, vamos escolher uma função de hash para o arquivo *conta* usando a chave de busca *nome_agência*. A função de hash que escolhemos precisa ter as propriedades desejáveis não apenas sobre o arquivo de exemplo *conta* que usamos, mas também sobre um arquivo *conta* do tamanho realista para um grande banco com muitas agências.

Suponha que decidimos ter 26 buckets e definimos uma função de hash que mapeia os nomes começando com a i -ésima letra do alfabeto para o i -ésimo bucket. A função de hash tem a virtude da simplicidade, mas não consegue oferecer uma distribuição uniforme, pois esperamos que mais nomes de agência comecem com as letras B e R, em vez de Q e X, por exemplo.

Agora suponha que queremos uma função de hash sobre a chave de busca *saldo*. Suponha que o saldo mínimo seja 1 e o saldo máximo seja 100.000 e usemos uma função de hash que divide os valores em 10 intervalos, 1-10.000, 10.001-20.000 e assim por diante. A distribuição dos valores de chave de busca é uniforme (pois cada bucket tem o mesmo número de diferentes valores de *saldo*), mas não é aleatória. Contudo, os registros com saldos entre 1 e 10.000 são muito mais comuns do que os registros com saldos entre 90.001 e 100.000. Como resultado, a distribuição de registros não é uniforme – alguns buckets recebem mais registros do que outros. Se a função tiver uma distribuição aleatória, mesmo que tais correlações nas chaves de busca, a aleatoriedade da distribuição tornará muito provável que todos os buckets tenham aproximadamente o mesmo número de registros, desde que cada chave de busca ocorra apenas em uma pequena fração dos registros. (Se uma única chave de busca ocorrer em uma grande fração dos registros, o bucket que a contém provavelmente terá mais registros do que outros buckets, independente da função de hash utilizada.)

As funções de hash típicas realizam cálculos sobre a representação de máquina binária interna dos caracteres na chave de busca. Uma função de hash simples desse tipo primeiro calcula a soma das representações binárias dos caracteres de uma chave, depois retorna o módulo da soma dos números dos buckets. A Figura 12.21 mostra a aplicação desse tipo de esquema, com 10 buckets, ao arquivo *conta*, sob a suposição de que a *i*-ésima letra do alfabeto seja representada pelo inteiro *i*.

A seguinte função de hash pode ser usada para desmembrar uma string em uma implementação Java:

$$s[0] * 31^{(n-1)} + s[1] * 31^{(n-2)} + \dots + s[n-1]$$

A função pode ser implementada de forma eficiente definindo o resultado do hash inicialmente como 0, e percorrendo do primeiro ao último caractere da string, em cada etapa multiplicando o valor de hash por 31 e depois somando o caractere seguinte (tratado como um inteiro). O resultado do módulo dessa função pelo número de buckets pode ser usado para a indexação.

bucket 0		

bucket 1		

bucket 2		

bucket 3		
A-217	Brighton	750
A-305	Round Hill	350

bucket 4		
A-222	Redwood	700

bucket 5		
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6		

bucket 7		
A-215	Mianus	700

bucket 8		
A-101	Downtown	500
A-110	Downtown	600

bucket 9		

Figura 12.21 Organização de hash do arquivo *conta*, com a chave *nome_agência*.

As funções de hash exigem um projeto cuidadoso. Uma função de hash errada pode resultar em um tempo de pesquisa proporcional ao número de chaves de busca no arquivo. Uma função bem projetada oferece um tempo de pesquisa médio constante (e pequeno), independente do número de chaves de busca no arquivo.

Tratamento de estouros de bucket

Até aqui, consideramos que, quando um registro é inserido, o bucket ao qual ele é mapeado possui espaço para armazenar o registro. Se o bucket não tiver espaço suficiente, acontece um **estouro de bucket**. O estouro de bucket pode ocorrer por vários motivos:

- **Buckets insuficientes.** O número de buckets, que indicamos com n_b , precisa ser escolhido de modo que $n_b > n/f_r$, onde n_r indica o número total de registros que serão armazenados e f_r indica o número de registros que caberão em um bucket. Essa designação, naturalmente, considera que o número total de registros é conhecido quando a função de hash é escolhida.
- **Distorção.** Alguns buckets recebem mais registros do que outros, de modo que um bucket pode estourar mesmo quando outros buckets ainda têm espaço. Essa situação é chamada de **distorção de bucket**. A distorção pode ocorrer por dois motivos:
 1. Vários registros podem ter a mesma chave de busca.
 2. A função de hash escolhida pode resultar na distribuição não uniforme das chaves de busca.

Para que a probabilidade de estouro de bucket seja reduzida, o número de buckets é escolhido como $(n/f_r)^*(1+d)$, onde d é um fator de fudge, normalmente em torno de 0,2. Algum espaço é desperdiçado: cerca de 20% do espaço nos baldes estará vazio. No entanto, o benefício é que a probabilidade de estouro é reduzida.

Apesar da alocação de mais alguns buckets do que o necessário, o estouro de bucket ainda pode ocorrer. Tratamos o estouro de bucket usando **buckets de estouro**. Se um registro tiver de ser inserido no bucket b , e b já estiver cheio, o sistema oferece um bucket de estouro para b e insere o registro no bucket de estouro. Se o bucket de estouro também estiver cheio, o sistema oferece outro bucket de estouro, e assim por diante. Todos os buckets de estouro de determinado bucket são encadeados em uma lista interligada, como na Figura 12.22. O tratamento do estouro usando tal lista interligada é chamado de **encadeamento de estouro**.

Temos de mudar o algoritmo de pesquisa ligeiramente para lidar com o encadeamento de estouro. Como antes, o sistema usa a função de hash sobre a chave de busca para identificar um bucket b . O sistema precisa examinar todos os registros no bucket b para ver se eles combinam com a chave de busca, como antes. Além disso, se o bucket b tiver buckets de estouro, o sistema também terá de examinar os registros em todos os buckets de estouro.

A forma da estrutura de hash que acabamos de descrever às vezes é chamada de **hashing fechado**. Sob uma técnica alternativa, chamada **hashing aberto**, o conjunto de buckets é fixo, e não existem cadeias de estouro. Em vez disso, se um bucket estiver cheio, o sistema insere registros em al-

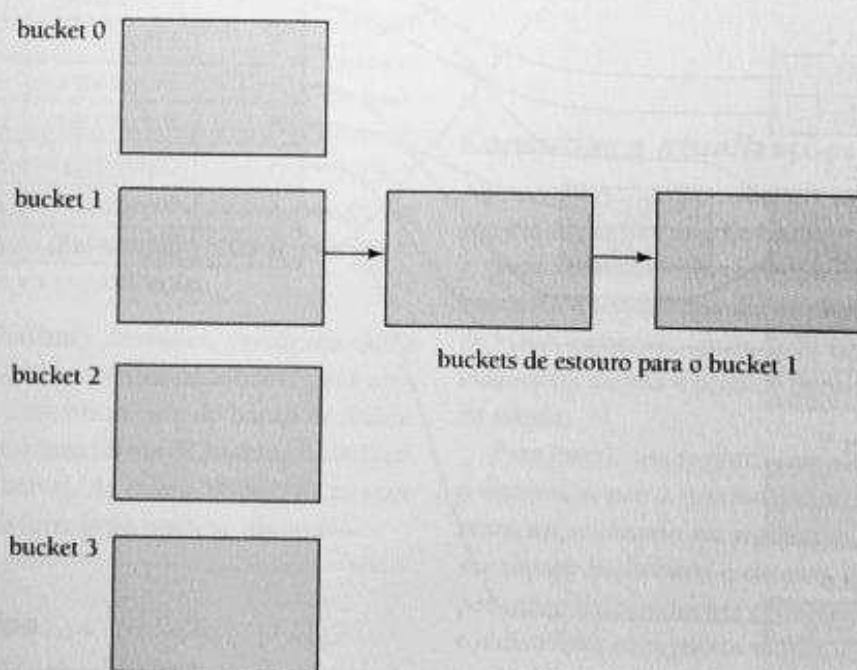


Figura 12.22 Encadeamento de estouro em uma estrutura de hash.

gum outro bucket no conjunto inicial de buckets B . Uma política é usar o próximo bucket (em ordem cíclica) que possui espaço. Essa política é chamada *sonda linear*. Outras políticas, como o cálculo de outras funções de hash, também são usadas. O hashing aberto tem sido usado para construir tabelas de símbolos para compiladores e assemblers, mas o hashing fechado é preferível para sistemas de banco de dados. O motivo é que a exclusão sob o hashing aberto é trabalhosa. Normalmente, compiladores e assemblers realizam apenas operações de pesquisa e inserção em suas tabelas de símbolos. Porém, em um sistema de banco de dados, é importante poder lidar com a exclusão tão bem quanto a inserção. Assim, o hashing aberto tem importância secundária na implementação do banco de dados.

Uma desvantagem importante da forma de hashing que descrevemos é que precisamos escolher a função de hash quando implementamos o sistema, e ela não pode ser mudada facilmente depois disso, se o arquivo sendo indexado crescer ou diminuir. Como a função h mapeia os valores de chave de busca a um conjunto fixo B dos endereços de bucket, desperdiçamos espaço se B se tornar grande para lidar com o crescimento futuro do arquivo. Se B for muito pequeno, os buckets contêm registros de muitos valores dife-

rentes de chave de busca, e pode haver estouros de bucket. Com o crescimento do arquivo, o desempenho sofre. Estudaremos depois, na seção “Hashing dinâmico”, como o número de buckets e a função de hash pode ser alterada dinamicamente.

Índices de hash

O hashing pode ser usado não apenas para organização de arquivo, mas também para a criação da estrutura de índice. Um índice de hash organiza as chaves de busca, com seus ponteiros associados em uma estrutura de arquivo de hash. Construimos um índice de hash da seguinte maneira. Apliquamos uma função de hash sobre uma chave de busca para identificar um bucket, e armazenamos a chave e seus ponteiros associados no bucket (ou em buckets de estouro). A Figura 12.23 mostra um índice de hash secundário sobre o arquivo *conta*, para a chave de busca *número_conta*. A função de hash na figura calcula a soma dos dígitos do número de conta módulo 7. O índice de hash possui sete buckets, cada um com tamanho 2 (índices realistas, naturalmente, teriam tamanhos de bucket muito maiores). Um dos buckets tem três chaves mapeadas, de modo que possui um

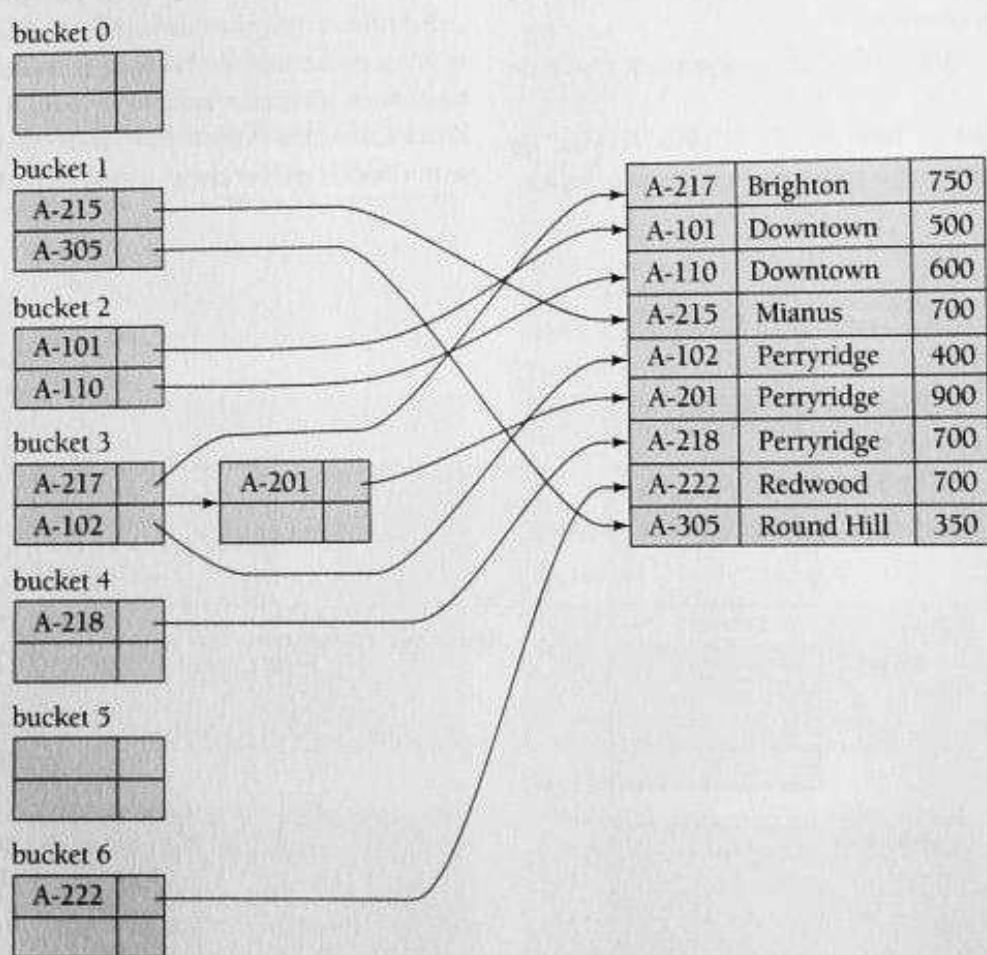


Figura 12.23 Índice de hash sobre a chave de busca *número_conta* do arquivo *conta*.

bucket de estouro. Nesse exemplo, *numero_conta* é uma chave primária para *conta*, de modo que cada chave de busca tem apenas um ponteiro associado. Em geral, vários ponteiros podem estar associados a cada chave.

Usamos o termo *índice de hash* para indicar as estruturas de arquivo além de índices de hash secundários. Estritamente falando, os índices de hash são apenas estruturas de índice secundárias. Um índice de hash nunca é necessário como uma estrutura de índice agrupado, pois, se um arquivo já estiver organizado por hashing, não será necessária uma estrutura de índice de hash separada sobre ele. No entanto, como a organização de arquivo de hash oferece o mesmo acesso direto aos registros que a indexação, fazemos de conta que um arquivo organizado por hashing também possui um índice de hash agrupado sobre ele.

Hashing dinâmico

Como já vimos, a necessidade de consertar o conjunto B de endereços de bucket apresenta um problema sério com a técnica de hashing estático da seção anterior. A maioria dos bancos de dados se torna muito grande com o tempo. Se tivermos de usar o hashing estático para tal banco de dados, temos três classes de opções:

1. Escolha uma função de hash com base no tamanho do arquivo atual. Essa opção resultará na diminuição de desempenho quando o banco de dados crescer.
2. Escolha uma função de hash com base no tamanho antecipado do arquivo em algum ponto no futuro. Embora a diminuição de desempenho seja evitada, uma quantidade significativa de espaço pode ser desperdiçada inicialmente.
3. Reorganize periodicamente a estrutura de hash em resposta ao crescimento do arquivo. Tal reorganização envolve escolha de uma nova função de hash, recalculando a função de hash em cada registro do arquivo, e gerando novas atribuições de bucket. Essa reorganização é uma operação maciça e demorada. Além do mais, é necessário proibir o acesso ao arquivo durante a reorganização.

Várias técnicas de hashing dinâmico permitem que a função de hash seja modificada dinamicamente para acomodar o crescimento ou encolhimento do banco de dados. Nesta seção, descrevemos uma forma de hashing dinâmico, chamada **hashing extensível**. As notas bibliográficas contêm referências a outras formas de hashing dinâmico.

Estrutura de dados

O hashing extensível lida com as mudanças no tamanho do banco de dados, dividindo e unindo os buckets enquanto o

banco de dados cresce e encurta. Como resultado, a eficiência do espaço é mantida. Além do mais, como a reorganização é realizada somente em um bucket de cada vez, a sobrecarga resultante no desempenho é aceitavelmente baixa.

Com o hashing extensível, escolhemos uma função de hash h com as propriedades desejáveis de uniformidade e aleatoriedade. Porem, essa função de hash gera valores por um intervalo relativamente grande – a saber, inteiros binários de b bits. Um valor típico de b é 32.

Não criamos um bucket para cada valor de hash. Na realidade, 2^{32} é mais de 4 bilhões, e essa quantidade é excessiva para quase tudo, menos para bancos de dados grandes. Em vez disso, criamos buckets por demanda, quando registros são inseridos no arquivo. Não usamos os b bits inteiros do valor de hash inicialmente. A qualquer ponto, usamos i bits, onde $0 \leq i \leq b$. Esses i bits são usados como um deslocamento para uma tabela adicional de endereços de bucket. O valor de i cresce e encurta com o tamanho do banco de dados.

A Figura 12.24 mostra uma estrutura geral de hash extensível. O i que aparece acima da tabela de endereços de bucket na figura indica que i bits do valor de hash $h(K)$ são necessários para determinar o bucket correto para K . Esse número, naturalmente, mudará quando o arquivo crescer. Embora i bits sejam necessários para encontrar a entrada correta na tabela de endereços de bucket, várias entradas de tabela consecutivas podem apontar para o mesmo bucket. Todas essas entradas terão um prefixo de hash comum, mas o tamanho desse prefixo pode ser menor que i . Portanto, associamos a cada bucket um inteiro dando o tamanho do prefixo de hash comum. Na Figura 12.24, o inteiro associado ao bucket j aparece como i_j . O número de entradas da tabela de endereços de bucket que apontam para o bucket j é

$$2^{(i-i_j)}$$

Consultas e atualizações

Agora veremos como realizar a pesquisa, inserção e exclusão em uma estrutura de hash extensível.

Para localizar o bucket contendo o valor de chave de busca K_i , o sistema apanha os primeiros i bits de alta ordem de $h(K_i)$, examina a entrada de tabela correspondente para essa string de bits e segue o ponteiro de bucket na entrada da tabela.

Para inserir um registro com valor de chave de busca K_i , o sistema segue o mesmo procedimento de antes, para a pesquisa, acabando em algum bucket – digamos, j . Se houver espaço no bucket, o sistema insere o registro nele. Se, por outro lado, o bucket estiver cheio, ele precisa dividi-lo e redistribuir os registros atuais, mais o novo. Para dividir o bucket, o sistema precisa primeiro determinar pelo valor de hash se ele precisa aumentar o número de bits utilizados.

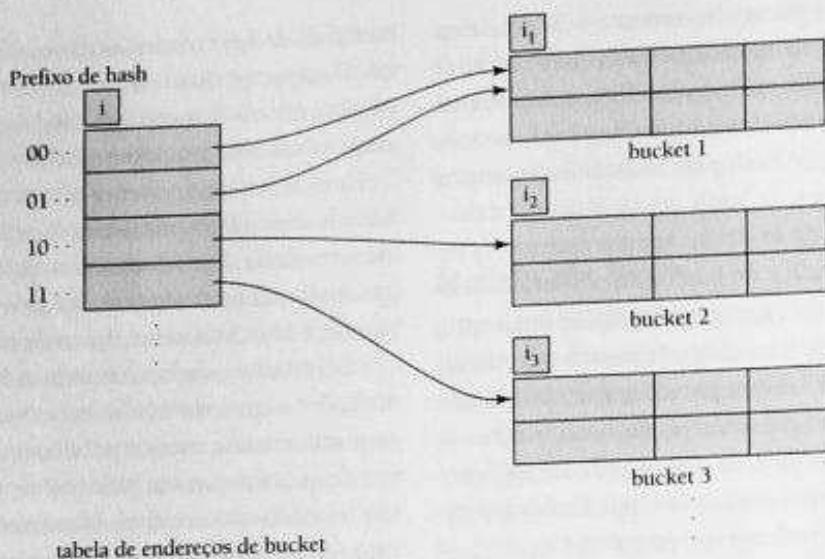


Figura 12.24 Estrutura de hash geral extensível.

- Se $i = i_j$, somente uma entrada na tabela de endereços de bucket aponta para o bucket j . Portanto, o sistema precisa aumentar o tamanho da tabela de endereços de bucket de modo que ela possa incluir ponteiros para os dois buckets que resultam da divisão do bucket j . Ele faz isso considerando um bit adicional do valor de hash. Ele incrementa o valor de i em 1, dobrando o tamanho da tabela de endereços de bucket. Ele substitui cada entrada por duas entradas, ambas contendo o mesmo ponteiro da entrada original. Agora, duas entradas na tabela de endereços de bucket apontam para o bucket j . O sistema aloca um novo bucket (bucket z) e define a segunda entrada para que aponte para o novo bucket. Ele define i_j e i_z como i . Em seguida, ele calcula novamente o hash de cada registro no bucket j e, dependendo dos i primeiros bits (lembre-se de que o sistema somou 1 a i), ele o mantém no bucket j ou o aloca ao bucket recém-criado.

O sistema agora tenta novamente inserir o novo registro. Normalmente, a tentativa terá sucesso. Porém, se todos os registros no bucket j , além do novo registro, tiverem o mesmo prefixo de valor de hash, será preciso dividir o bucket novamente, pois todos os registros no bucket j e o novo registro são atribuídos ao mesmo bucket. Se a função de hash tiver sido cuidadosamente escolhida, é pouco provável que uma única inserção exija que o bucket seja dividido mais de uma vez, a menos que haja uma grande quantidade de registros com a mesma chave de busca. Se todos os registros no bucket j tiverem o mesmo valor de chave de busca, nenhuma quantidade de divisão ajudará. Nesses casos, os buckets de estouro são usados para armazenar os registros, como no hashing estático.

- Se $i > i_j$, então mais de uma entrada na tabela de endereços de bucket aponta para o bucket j . Assim, o sistema pode dividir o bucket j sem aumentar o tamanho da tabela de

endereços de bucket. Observe que todas as entradas que apontam para o bucket j correspondem a prefixos de hash que têm o mesmo valor nos i_j bits mais à esquerda. O sistema aloca um novo bucket (bucket z) e define i_j e i_z com o valor que resulta da soma de 1 ao valor original de i_j . Em seguida, o sistema precisa ajustar as entradas na tabela de endereços de bucket que anteriormente apontava para o bucket j . (Observe que, com o novo valor para i_j , nem todas as entradas correspondem aos prefixos de hash que têm o mesmo valor nos i_j bits mais à esquerda.) O sistema deixa a primeira metade das entradas como estavam (apontando para o bucket j) e define todas as entradas restantes para que apontem para o bucket recém-criado (bucket z). Em seguida, como no caso anterior, o sistema recalcula o hash de cada registro no bucket j e o aloca ao bucket j ou ao bucket z recém-criado.

O sistema, então, tenta inserir novamente. No caso improvável de nova falha, ele aplica um dos dois casos, $i = i_j$ ou $i > i_j$, como for mais apropriado.

Observe que, nos dois casos, o sistema precisa recalcular a função de hash apenas sobre os registros do bucket j .

Para excluir um registro com valor de chave de busca K_i , o sistema segue o mesmo procedimento de pesquisa anterior, acabando em algum bucket – digamos, j . Ele remove a chave de busca do bucket e o registro do arquivo. O bucket também é removido se ficar vazio. Observe que, nesse ponto, vários buckets podem ser unidos, e o tamanho da tabela de endereços de bucket pode ser reduzido ao meio. O procedimento para decidir quais buckets podem ser unidos e como uni-los fica para você como um exercício. As condições sob as quais a tabela de endereços de bucket pode ter o tamanho reduzido também ficam como um exercício. Ao contrário da união de buckets, a mudança do tamanho da

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Figura 12.25 Exemplo de arquivo de conta.

tabela de endereços de bucket é uma operação um tanto dispendiosa se a tabela for grande. Portanto, pode valer a pena reduzir o tamanho da tabela de endereços de bucket somente se o número de buckets reduzir muito.

Nosso arquivo de exemplo *conta* da Figura 12.25 ilustra a operação de inserção. Os valores de hash de 32 bits sobre *nome_agência* aparecem na Figura 12.26. Suponha que, inicialmente, o arquivo esteja vazio, como na Figura 12.27. Inserimos os registros um por um. Para ilustrar todos os recursos do hashing extensível em uma estrutura pequena, faremos a suposição não realista de que um bucket só pode manter dois registros.

Inserimos o registro (A-217, Brighton, 750). A tabela de endereços de bucket contém um ponteiro para um bucket, e o sistema insere o registro. Em seguida, inserimos o registro (A-101, Downtown, 500). O sistema também coloca esse registro no único bucket de nossa estrutura.

Quando tentamos inserir o próximo registro (A-110, Downtown, 600), descobrimos que o bucket está cheio. Como $i = i_0$, precisamos aumentar o número de bits que usamos a partir do valor de hash. Agora usamos 1 bit, permitindo-nos $2^1 = 2$ buckets. Esse aumento no número de bits exige dobrar o tamanho da tabela de endereços de bucket para duas entradas. O sistema divide o bucket, colocando no novo bucket aqueles registros cuja chave de busca possui um valor de hash começando com 1, e deixando no bucket original os outros registros. A Figura 12.28 mostra o estado de nossa estrutura após a divisão.

Em seguida, inserimos (A-215, Mianus, 700). Como o primeiro bit de $h(\text{Mianus})$ é 1, temos de inserir esse registro no bucket apontado pela entrada “1” na tabela de endereços de bucket. Mais uma vez, encontramos o bucket cheio e $i = i_1$. Aumentamos o número de bits que usamos a partir do hash para 2. Esse aumento no número de bits exige a dupli-

<i>nome_agência</i>	$h(\text{nome_agência})$
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001

Figura 12.26 Função de hash para *nome_agência*.

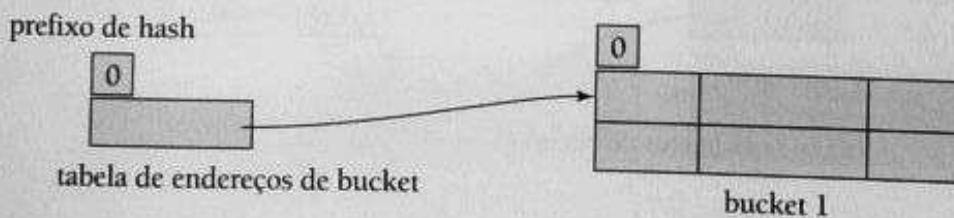


Figura 12.27 Estrutura de hash extensível inicial.

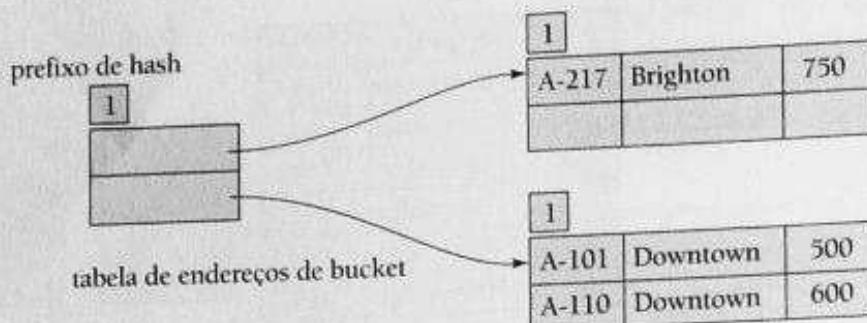


Figura 12.28 Estrutura de hash após três inserções.

cação do tamanho da tabela de endereços de bucket para quatro entradas, como na Figura 12.29. Como o bucket da Figura 12.28 para o prefixo de hash 0 não foi dividido, as duas entradas da tabela de endereços de bucket de 00 e 01 apontam para esse bucket.

Para cada registro no bucket da Figura 12.28 para o prefixo de hash 1 (o bucket sendo dividido), o sistema examina os 2 primeiros bits do valor de hash para determinar qual bucket da nova estrutura deverá mantê-lo.

Em seguida, inserimos (A-102, Perryridge, 400), que entra no mesmo bucket de Mianus. A inserção a seguir, de (A-201, Perryridge, 900), resulta em um estouro de bucket, levando a um aumento no número de bits e a duplicação do tamanho da tabela de endereços de bucket. A inserção do terceiro registro de Perryridge, (A-218, Perryridge, 700), leva a outro estouro. Porem, esse estouro não pode ser tratado pelo aumento do número de bits, pois existem três registros exatamente com o mesmo valor de hash. Logo, o sistema usa um bucket de estouro, como na Figura 12.30.

Continuamos dessa maneira até termos inserido todos os registros de conta da Figura 12.25. A estrutura resultante aparece na Figura 12.31.

Hashing estático versus hashing dinâmico

Agora, examinamos as vantagens e desvantagens do hashing extensível, em comparação com o hashing estático. A vantagem principal do hashing extensível é que o desempenho não diminui quando o arquivo aumenta. Além do mais, existe uma sobrecarga de espaço mínima. Embora a tabela de endereços de bucket gere uma sobrecarga adicional, ela contém um ponteiro para cada valor de hash para o tamanho de prefixo atual. Essa tabela, portanto, é pequena. A principal economia de espaço do hashing extensível em relação a outras formas de hashing é que nenhum bucket precisa ser reservado para crescimento futuro; em vez disso, os buckets podem ser alocados dinamicamente.

Uma desvantagem do hashing extensível é que a pesquisa envolve um nível de indireção adicional, pois o sistema precisa acessar a tabela de endereços de bucket antes de acessar o próprio bucket. Essa referência extra possui apenas um efeito secundário sobre o desempenho. Embora as estruturas de hash que discutimos na seção "Hashing estático" não tenham esse nível de indireção extra, elas perdem sua vantagem secundária no desempenho quando se tornam cheias.

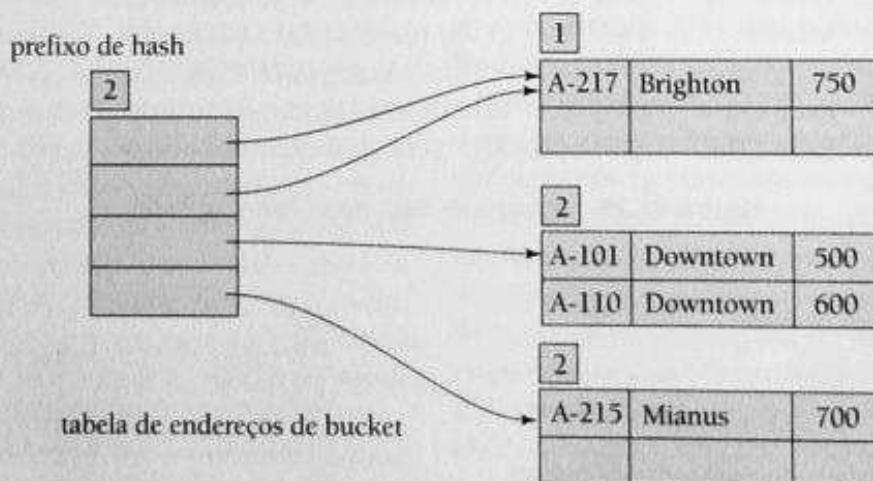


Figura 12.29 Estrutura de hash após quatro inserções.

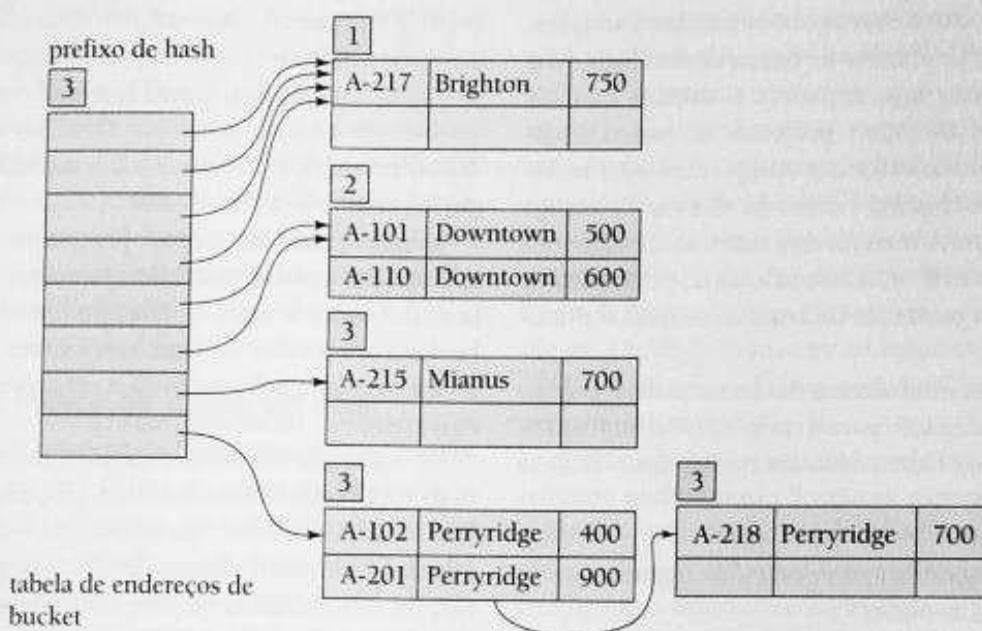


Figura 12.30 Estrutura de hash após sete inserções.

Assim, o hashing extensível parece ser uma técnica altamente atraente, desde que estejamos dispostos a aceitar a complexidade adicional envolvida em sua implementação. As notas bibliográficas referenciam descrições mais detalhadas da implementação de hashing extensível.

As notas bibliográficas também oferecem referências a outra forma de hashing dinâmico, chamado hashing linear, que evita o nível de indireção extra associado ao hashing extensível, ao custo possível de mais buckets de estouro.

Comparação de indexação ordenada e hashing

Vimos vários esquemas de indexação ordenada e vários esquemas de hashing. Podemos organizar arquivos de registros como arquivos ordenados, usando a organização sequencial indexada ou organizações de árvore B*. Como alternativa, podemos organizar os arquivos usando o hashing. Finalmente, podemos organizar os como arquivos de heap, em que os registros não são ordenados de qualquer modo em particular.

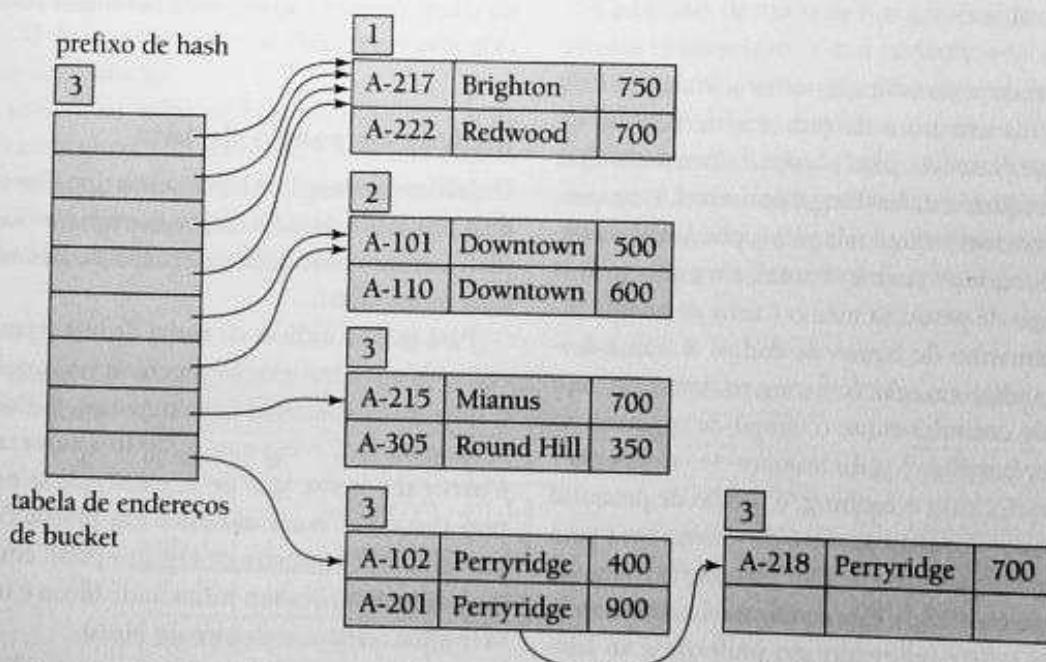


Figura 12.31 Estrutura de hash extensível para o arquivo conta.

Cada esquema possui vantagens em certas situações. Um implementador de sistema de banco de dados poderia oferecer muitos esquemas, deixando a decisão final de quais esquemas utilizar para o projetista do banco de dados. Porém, essa técnica exige que o implementador escreva mais código, aumentando o custo do sistema e o espaço que o sistema ocupa. A maioria dos sistemas de banco de dados admite árvores B* e, adicionalmente, pode admitir alguma forma de organização de arquivo de hash ou índices de hash.

Para fazer uma escolha sensata das técnicas de organização de arquivo e indexação para uma relação, o implementador ou projetista de banco de dados precisa considerar as seguintes questões:

- O custo da reorganização periódica da organização de índice ou hash é aceitável?
- Qual é a frequência relativa de inserção ou exclusão?
- É desejável otimizar o tempo médio de acesso às custas de aumentar o tempo de acesso no pior caso?
- Que tipos de consultas os usuários provavelmente realizam?

Já examinamos as três primeiras questões, primeiro em nossa revisão dos méritos relativos de técnicas de indexação específicas, e novamente em nossa discussão sobre técnicas de hashing. A quarta questão, o tipo de consulta esperado, é critica para a escolha da indexação ordenada ou hashing.

Se a maioria das consultas for da forma

```
select A1, A2, ..., An
from r
where Ai = c
```

então, para processar a consulta, o sistema realizará uma pesquisa sobre uma estrutura de índice ordenado ou de hashing para os atributos A_i para o valor c. Para consultas dessa forma, um esquema de hashing é preferível. Uma pesquisa de índice ordenado exige tempo proporcional ao log do número de valores em r para A_i. Porem, em uma estrutura de hash, o tempo de pesquisa médio é uma constante independente do tamanho do banco de dados. A única desvantagem de um índice em relação a uma estrutura de hash para essa forma de consulta é que o tempo de pesquisa no pior caso é proporcional ao log do número de valores em r para A_i. Ao contrário, para o hashing, o tempo de pesquisa no pior caso é proporcional ao número de valores em r para A_i. Porem, o tempo de pesquisa no pior caso provavelmente não ocorrerá com o hashing, e este é preferível nesse caso.

As técnicas de índice ordenado são preferíveis ao hashing em casos em que a consulta especifica um intervalo de valores. Essa consulta tem a seguinte forma:

```
select A1, A2, ..., An
from r
where Ai ≤ c2 and Ai ≥ c1
```

Em outras palavras, a consulta anterior localiza todos os registros com valores de A_i entre c₁ e c₂.

Vamos considerar como processamos essa consulta usando um índice ordenado. Primeiro, realizamos uma pesquisa sobre o valor c₁. Quando tivermos encontrado o bucket para o valor c₁, seguimos a cadeia de ponteiros no índice para ler o próximo bucket em ordem, e continuamos dessa maneira até alcançarmos c₂.

Se, em vez de um índice ordenado, tivermos uma estrutura de hash, podemos realizar uma pesquisa sobre c₁ e podemos localizar o bucket correspondente – mas, em geral, não é fácil determinar o próximo bucket que precisa ser examinado. A dificuldade surge porque uma boa função de hash atribui valores aleatoriamente aos buckets. Assim, não existe uma noção simples de “próximo bucket na ordem classificada”. O motivo de não podermos encadear buckets em ordem classificada sobre A_i é que cada bucket recebe muitos valores de chave de busca. Como os valores são espalhados aleatoriamente pela função de hash, os valores no intervalo especificado provavelmente estarão espalhados por muitos ou todos os buckets. Portanto, temos de ler todos os buckets para encontrar as chaves de busca solicitadas.

Normalmente, projetista escolherá a indexação ordenada, a menos que se saiba com antecedência que as consultas de intervalo serão pouco frequentes, quando o hashing seria escolhido. As organizações de hash são particularmente úteis para arquivos temporários criados durante o processamento da consulta, se as pesquisas baseadas em um valor de chave forem exigidas, mas nenhuma consulta de intervalo será realizada.

Índices de mapa de bits

Os índices de mapa de bits são um tipo especializado de índice projetado para facilitar a consulta sobre chaves múltiplas, embora cada índice de mapa de bits seja baseado em uma única chave.

Para que os índices de mapa de bits sejam utilizados, os registros em uma relação precisam ser numerados seqüencialmente, começando com, digamos, 0. Dado um número n, deve ser fácil recuperar o registro numerado com n. Isso é particularmente fácil de se conseguir se os registros tiverem tamanho fixo e alocados em blocos consecutivos de um arquivo. O número de registro pode, então, ser traduzido facilmente para um número de bloco e um número que identifica o registro dentro do bloco.

Considere uma relação r, com um atributo A que pode assumir apenas um dentre um pequeno número de valores

(por exemplo, 2 a 20). Por exemplo, uma relação *info_cliente* pode ter um atributo *sexo*, que pode assumir apenas os valores m (masculino) ou f (feminino). Outro exemplo seria um atributo *nível_renda*, em que a renda foi dividida em 5 níveis: L1: 0-9.999, L2: 10.000-19.000, L3: 20.00-39.999, L4: 40.000-74.999 e L5: 75.000-∞. Aqui, os dados brutos podem assumir muitos valores, mas um analista de dados dividiu os valores em um pequeno número de intervalos para simplificar a análise desses dados.

Estrutura de índice de mapa de bits

Um mapa de bits é simplesmente um array de bits. Em sua forma mais simples, um índice de mapa de bits sobre o atributo *A* da relação *r* consiste em um mapa de bits para cada valor que *A* pode assumir. Cada mapa de bits possui tantos bits quanto o número de registros na relação. O *i*-ésimo bit do mapa de bits para o valor *v_i* é definido como 1 se o registro numerado com *i* tiver o valor *v_i* para o atributo *A*. Todos os outros bits do mapa de bits são definidos como 0.

Em nosso exemplo, existe um mapa de bits para o valor m e um para f. O *i*-ésimo bit do mapa de bits para m é definido como 1 se o valor de sexo do registro numerado com *i* for m. Todos os outros bits do mapa de bits para m são definidos como 0. De modo semelhante, o mapa de bits para f tem o valor 1 para os bits correspondentes aos registros com o valor f para o atributo *sexo*. Todos os outros bits têm o valor 0. A Figura 12.32 mostra um exemplo dos índices de mapa de bits em uma relação *info_cliente*.

Agora, vejamos quando os mapas de bit são úteis. A forma mais simples de obter todos os registros com valor m (ou valor f) seria simplesmente ler todos os registros da relação e selecionar aqueles registros com valor m (ou f, respectivamente). O índice do mapa de bits não ajuda realmente a agilizar essa seleção.

De fato, os índices de mapa de bits são úteis para seleções principalmente quando existem seleções sobre chaves múltiplas. Suponha que criemos um índice de mapa de bits sobre o atributo *nível_renda*, que descrevemos anteriormente, além do índice de mapa de bits sobre *sexo*.

número de registro	nome	sexo	endereço	nível_renda
0	John	m	Perryridge	L1
1	Diana	f	Brooklyn	L2
2	Mary	f	Jonestown	L1
3	Peter	m	Brooklyn	L4
4	Kathy	f	Perryridge	L3

Considere agora uma consulta que seleciona mulheres com renda no intervalo de 10.000-19.999. Essa consulta pode ser expressa como $\sigma_{\text{sexo}=f \wedge \text{nível_renda}=\text{L2}}(r)$. Para avaliar essa seleção, apanhamos os mapas de bits para o valor f de sexo e o mapa de bits para o valor L2 de *nível_renda*, e realizamos uma interseção (and lógico) dos dois mapas de bits. Em outras palavras, calculamos um novo mapa de bits em que o bit *i* tem valor 1 se o *i*-ésimo bit dos dois mapas de bits forem 1; caso contrário, ele tem um valor 0. No exemplo da Figura 12.32, a interseção do mapa de bits para *sexo = f* (01101) e o mapa de bits para *nível_renda = L2* (01000) gera o mapa de bits 01000.

Como o primeiro atributo pode assumir 2 valores, e o segundo pode assumir 5 valores, esperaríamos que apenas cerca de 1 em 10 registros, na média, satisfaça uma condição combinada sobre os dois atributos. Se houver outras condições, a fração dos registros que satisfazem todas as condições provavelmente será muito pequena. O sistema pode, então, calcular o resultado da consulta localizando todos os bits com valor 1 no mapa de bits da interseção e apanhando os registros correspondentes. Se a fração for grande, a varredura da relação inteira continuaria sendo a alternativa mais barata.

Outro uso importante dos mapas de bits é contar o número de tuplas satisfazendo determinada seleção. Essas consultas são importantes para a análise de dados. Por exemplo, se quisermos descobrir quantas mulheres possuem um nível de renda L2, calculamos a interseção dos dois mapas de bits e depois contamos o número de bits que são 1 no mapa de bits da interseção. Assim, podemos chegar ao resultado a partir do índice de mapa de bits, sem sequer acessar a relação.

Os índices de mapa de bits geralmente são muito pequenos em comparação com o tamanho real da relação. Os registros normalmente têm pelo menos dezenas de bytes a centenas de bytes, enquanto um único bit representa um registro em um mapa de bits. Assim, o espaço ocupado por um único mapa de bits normalmente é menor do que 1% do espaço ocupado pela relação. Por exemplo, se o tamanho do registro para determinada relação for de 100 bytes, en-

Mapas de bits para sexo		Mapas de bits para nível_renda	
m	10010	L1	10100
f	01101	L2	01000
		L3	00001
		L4	00010
		L5	00000

Figura 12.32 Índices de mapa de bits na relação *info_cliente*.

tão o espaço ocupado por um único mapa de bits seria 1/8 do 1% do espaço ocupado pela relação. Se um atributo A da relação puder assumir apenas um dentre 8 valores, um índice de mapa de bits sobre o atributo A consistiria em 8 mapas de bits, que juntos ocupariam apenas 1% do tamanho da relação.

A exclusão de registros cria lacunas na sequência de registros, pois o deslocamento de registros (ou números de registro) para preencher as lacunas seria extremamente dispendioso. Para reconhecer os registros excluídos, podemos armazenar um mapa de bits de existência, em que o bit i é 0 se o registro i não existir, e 1 se ele existir. Veremos a necessidade da existência de mapas de bits na próxima seção. A inserção de registros não deverá afetar a numeração de sequência de outros registros. Portanto, podemos realizar a inserção acrescentando registros ao final do arquivo ou substituindo os registros excluídos.

Implementação eficiente de operações de mapa de bits

Podemos calcular a interseção dos dois mapas de bits com facilidade usando um loop for: a i -ésima iteração do loop calcula o and dos i -ésimos bits dos dois mapas de bits. Podemos agilizar bastante o cálculo da interseção usando instruções and bit a bit, aceitas pela maioria dos conjuntos de instruções de computador. Uma palavra (*word*) normalmente consiste em 32 ou 64 bits, dependendo da arquitetura do computador. Uma instrução and bit a bit apanha duas palavras como entrada e gera uma palavra em que cada bit é o and lógico dos bits nas posições correspondentes das palavras de entrada. O que é importante observar é que uma única instrução and bit a bit pode calcular a interseção de 32 ou 64 bits ao mesmo tempo.

Se uma relação tivesse 1 milhão de registros, cada mapa de bits teria 1 milhão de bits, ou 128 kilobytes. Apenas 31.250 instruções são necessárias para calcular a interseção dos dois mapas de bits para nossa relação, considerando um tamanho de palavra de 32 bits. Assim, o cálculo das interseções de mapa de bits é uma operação extremamente rápida.

Assim como a interseção de mapa de bits é útil para calcular o and de duas condições, a união de mapa de bits é útil para calcular o or de duas condições. O procedimento para a união de mapa de bits é exatamente o mesmo da interseção, exceto que usamos instruções or bit a bit em vez de instruções and bit a bit.

A operação de complemento pode ser usada para calcular um predicado envolvendo a negação de uma condição, como not (nivel_renda = L1). O complemento de um mapa de bits é gerado pela complementação de cada bit do mapa de bits (o complemento de 1 é 0, e o complemento de 0 é 1). Pode parecer que not (nivel_renda = L1) pode ser implemen-

tado apenas calculando-se o complemento do mapa de bits para o nível de renda L1. Porém, se alguns registros tiverem sido excluídos, simplesmente calcular o complemento de um mapa de bits não é suficiente. Os bits correspondentes a tais registros seriam 0 no mapa de bits original, mas se tornariam 1 no complemento, embora os registros não existam. Um problema semelhante também surge quando o valor de um atributo é nulo. Por exemplo, se o valor de nivel_renda fosse nulo, o bit seria 0 no mapa de bits original para o valor L1, e 1 no mapa de bits do complemento.

Para ter certeza de que os bits correspondentes aos registros excluídos sejam definidos como 0 no resultado, o mapa de bits do complemento precisa passar pela interseção com o mapa de bits de existência, de modo a desativar os bits para os registros excluídos. De modo semelhante, para lidar com valores nulos, o mapa de bits de complemento também precisa sofrer uma interseção com o complemento do mapa de bits para o valor nulo.¹

A contagem do número de bits que são 1 em um mapa de bits pode ser feita rapidamente por uma técnica inteligente. Podemos manter um array com 256 entradas, em que a i -ésima entrada armazena o número de bits que são 1 na representação binária de i . Defina a contagem total inicialmente como 0. Apanhamos cada byte do mapa de bits, o usamos para indexar esse array e somamos a contagem armazenada à contagem total. O número de operações de adição seria 1/8 do número de tuplas, e assim o processo de contagem é muito eficiente. Um array grande (usando $2^{16} = 65.536$ entradas), indexado por pares de bytes, ocasionaria um aumento de velocidade ainda maior, mas a um custo mais alto no armazenamento.

Mapas de bits e árvores B⁺

Os mapas de bits podem ser combinados com os índices normais de árvore B⁺ para relações em que alguns valores de atributo são extremamente comuns, e outros valores também ocorrem, mas com muito menos freqüência. Em uma folha de índice de árvore B⁺, para cada valor, normalmente manteríamos uma lista de todos os registros com esse valor para o atributo indexado. Cada elemento da lista seria um identificador de registro, consistindo em pelo menos 32 bits, e normalmente mais. Para um valor que ocorre em muitos registros, armazenamos um mapa de bits no lugar de uma lista de registros.

Suponha que determinado valor v_i ocorra em 1/16 dos registros em uma relação. Considere que N é o número de registros na relação, e suponha que um registro tenha um

¹ O tratamento de predicados como é desconhecido causaria outras complicações, que em geral exigiriam o uso de um mapa de bits extra para acompanhar quais resultados de operação são desconhecidos.

número de 64 bits que o identifica. O mapa de bits só precisa de 1 bit por registro, ou N bits no total. Ao contrário, a representação de lista exige 64 bits por registro em que o valor ocorre, ou $64 * N/16 = 4N$ bits. Assim, um mapa de bits é preferível para representar a lista de registros para o valor v. Em nosso exemplo (com um identificador de registro de 64 bits), se menos de 1 em 64 registros tiverem determinado valor, a representação de lista é preferível para identificar registros com esse valor, pois utiliza menos bits do que a representação de mapa de bits. Se mais de 1 em 64 registros tiverem esse valor, a representação de mapa de bits será preferível.

Assim, os mapas de bits podem ser usados como um mecanismo de armazenamento compactado nos nós de folha das árvores B*, para os valores que ocorrem com muita frequência.

Definição de índice na SQL

O padrão SQL não oferece um meio para o usuário ou administrador do banco de dados controlar quais índices são criados e mantidos no sistema de banco de dados. Os índices não são necessários para exatidão, pois são estruturas de dados redundantes. Porém, os índices são importantes para o processamento eficiente de transações, incluindo transações e consultas de atualização. Os índices também são importantes para a imposição eficiente das restrições de integridade. Por exemplo, as implementações típicas impõem uma declaração de chave (Capítulo 4) criando um índice com a chave declarada como chave de busca do índice.

Em geral, um sistema de banco de dados pode decidir automaticamente quais índices devem ser criados. Porém, devido ao custo de espaço dos índices, bem como o efeito dos índices sobre o processamento de atualização, não é fácil fazer automaticamente as escolhas certas sobre quais índices manter. Portanto, a maioria das implementações da SQL oferece ao programador o controle sobre a criação e a remoção de índices via comandos da linguagem de definição de dados.

Ilustramos a sintaxe desses comandos sem seguida. Embora a sintaxe apresentada seja muito usada e aceita por muitos sistemas de banco de dados, ela não faz parte do padrão SQL-1999. Os padrões da SQL (até o SQL-1999, pelo menos) não admitem o controle do esquema do banco de dados, e se restringiram ao esquema lógico do banco de dados.

Criamos um índice com o comando `create index`, que tem o seguinte formato:

```
create index <nome-índice> on  
<nome-relação> (<lista-atributos>)
```

A *lista-atributos* é uma lista dos atributos das relações que formam a chave de busca para o índice.

Para definir um nome de índice *índice_agência* sobre a relação *agência* com *nome_agência* como chave de busca, escrevemos:

```
create index índice_agência on agência  
(nome_agência)
```

Se quisermos declarar que a chave de busca é uma chave candidata, acrescentamos o atributo `unique` à definição do índice. Assim, o comando

```
create unique index índice_agência on agência  
(nome_agência)
```

declara *nome_agência* como uma chave candidata para *agência*. Se, no momento em que entrarmos com o comando `create unique index`, *nome_agência* não for uma chave candidata, o sistema mostrará uma mensagem de erro, e a tentativa de criar o índice falhará. Se a tentativa de criação de índice tiver sucesso, qualquer tentativa subsequente para inserir uma tupla que infrinja a declaração de chave falhará. Observe que o recurso `unique` é redundante se o sistema de banco de dados admitir a declaração `unique` do padrão SQL.

Muitos sistemas de banco de dados também oferecem um meio de especificar o tipo de índice a ser usado (como árvore B* ou hashing). Alguns sistemas de banco de dados também permitem que um dos índices em uma relação seja declarado como agrupado; o sistema, então, armazena a relação classificada pela chave de busca do índice agrupado.

O nome do índice que especificamos para um índice é necessário para a remoção de um índice. O comando `drop index` tem este formato:

```
drop index <nome-índice>
```

Resumo

- Muitas consultas referenciam apenas uma pequena proporção dos registros em um arquivo. Para reduzir a sobrecarga na pesquisa por esses registros, podemos construir índices para os arquivos que armazenam o banco de dados.
- Os arquivos sequenciais indexados são um dos esquemas de índice mais antigos usados nos sistemas de banco de dados. Para permitir a rápida recuperação de registros na ordem da chave de busca, os registros são armazenados seqüencialmente, e aqueles fora de ordem são encadeados. Para permitir o acesso aleatório rápido, usamos uma estrutura de índice.
- Existem dois tipos de índices que podemos usar: índices densos e índices esparsos. Os índices densos contêm en-

tradas para cada valor de chave de busca, enquanto os índices esparsos contêm entradas somente para os valores da chave de busca.

- Se a ordem de classificação de uma chave de busca combinar com a ordem de classificação de uma relação um índice na chave de busca é chamado de *índice agrupado*. Os outros índices são denominados *índices não agrupados ou secundários*. Os índices secundários melhoram o desempenho das consultas que usam chaves de busca diferentes da chave de busca para o índice agrupado. Porém, eles impõem uma sobrecarga na modificação do banco de dados.
- A principal desvantagem da organização de arquivo sequencial indexada é que o desempenho diminui à medida que o arquivo cresce. Para contornar essa deficiência, podemos usar um *índice de árvore B**.
- Um índice de árvore B* tem a forma de uma árvore *balanceada*, em que cada caminho da raiz da árvore até uma folha da árvore tem o mesmo tamanho. A altura de uma árvore B* é proporcional ao logaritmo de base N do número de registros na relação, em que cada nó não-folha armazena N ponteiros; o valor de N normalmente fica em torno de 50 a 100. As árvores B* são muito mais curtas do que outras estruturas de árvore binária balanceada, como árvores AVL, e por isso exigem menos acesso ao disco para localizar registros.
- A pesquisa nas árvores B* são diretas e eficientes. Porém, inserção e exclusão são um pouco mais complicadas, mas ainda eficientes. O número de operações exigidas para pesquisa, inserção e exclusão nas árvores B* é proporcional ao logaritmo de base N do número de registros na relação, em que cada nó não de folha armazena N ponteiros.
- Podemos usar árvores B* para indexar um arquivo contendo registros, além de organizar registros em um arquivo.
- Índices de árvore B são semelhantes aos índices de árvore B*. A principal vantagem de uma árvore B é que a árvore B elimina o armazenamento redundante dos valores de chave de busca. As principais desvantagens são a complexidade geral e o fanout reduzido para determinando tamanho de nó. Os projetistas de sistemas quase unanimemente preferem os índices de árvore B* aos índices de árvore B na prática.
- Organizações de arquivo sequenciais exigem uma estrutura de índice para localizar dados. As organizações de arquivo baseadas em hashing, ao contrário, nos permitem localizar o endereço de um item de dados diretamente pelo cálculo de uma função sobre o valor de chave de busca do registro desejado. Como não sabemos durante o projeto exatamente quais valores de chave de busca serão armazenados no arquivo, uma boa função

de hash para escolher é aquela que atribui valores de chave de busca aos buckets de modo que a distribuição seja uniforme e aleatória.

- O hashing estático utiliza funções de hash em que o conjunto de endereços de bucket é fixo. Essas funções de hash não podem acomodar com facilidade bancos de dados que se tornam muito grandes com o tempo. Existem várias técnicas de hashing dinâmico, que permitem a modificação da função de hash. Um exemplo é o hashing extensível, que lida com as mudanças no tamanho do banco de dados pela divisão e união de buckets à medida que o banco de dados cresce e encurta.
- Também podemos usar o hashing para criar índices secundários, esses índices são chamados *índices de hash*. Por conveniência de notação, assumimos que as organizações de arquivo possuem um índice de hash implícito sobre a chave de busca usada para o hashing.
- Índices ordenados, como árvores B* e índices de hash, podem ser usados para seleções com base em condições de igualdade envolvendo atributos isolados. Quando vários atributos estão envolvidos em uma condição de seleção, podemos realizar a interseção de identificadores de registro apanhados de vários índices.
- Os índices de mapa de bits oferecem uma representação bastante compacta para atributos de indexação com valores muito pouco distintos. As operações de interseção são extremamente rápidas nos mapas de bits, tornando-as ideais para o suporte a consultas sobre atributos múltiplos.

Termos de revisão

- Tipos de acesso
- Tempo de acesso
- Tempo de inserção
- Tempo de exclusão
- Sobrecarga de espaço
- Índice ordenado
- Índice agrupado
- Índice primário
- Índice não agrupado
- Índice secundário
- Arquivo sequencial indexado
- Registro/entrada de índice
- Índice denso
- Índice esparsa
- Índice multinível
- Chave composta
- Varredura sequencial
- Índice de árvore B*
- Árvore balanceada
- Organização de arquivo de árvore B*

- Índice de árvore B
- Hashing estático
- Organização de arquivo de hash
- Índice de hash
- Bucket
- Função de hash
- Estouro de bucket
- Distorção
- Hashing fechado
- Hashing dinâmico
- Hashing extensível
- Acesso por chave multipla
- Índices sobre chaves multiplas
- Índice de mapa de bits
- Operações de mapa de bits
 - Interseção
 - União
 - Complemento
 - Mapa de bits de existência

Exercícios práticos

12.1 Alguns índices agilizam o processamento da consulta; por que eles não poderiam ser mantidos em várias chaves de busca? Liste o máximo de respostas possível.

12.2 É possível, em geral, ter dois índices agrupados na mesma relação para diferentes chaves de busca? Explique sua resposta.

12.3 Construa uma árvore B* para o seguinte conjunto de valores de chave:

(2, 3, 5, 7, 11, 17, 19, 23, 29, 31)

Suponha que a árvore esteja inicialmente vazia e os valores sejam acrescentados em ordem crescente. Construa árvores B* para os casos em que o número de ponteiros que caberão em um nó é o seguinte:

- Quatro
- Seis
- Oito

12.4 Para cada árvore B* do Exercício 12.3, mostre a forma da árvore após cada uma das seguintes séries de operações:

- Inserir 9.
- Inserir 10.
- Inserir 8.
- Excluir 23.
- Excluir 19.

12.5 Considere o esquema de redistribuição modificado para as árvores B* descritas na seção "Organização de arquivos de árvore B*". Qual é a altura esperada da árvore como uma função de n ?

- 12.6 Repita o Exercício prático 12.3 para uma árvore B.
- 12.7 Suponha que estejamos usando o hashing extensível sobre um arquivo que contém registros com os seguintes valores de chave de busca:

2, 3, 5, 7, 11, 17, 19, 23, 29, 31

Mostre a estrutura de hash extensível para esse arquivo se a função de hash for $h(x) = x \bmod 8$ e os buckets puderem manter três registros.

- 12.8 Mostre como a estrutura de hash extensível do Exercício prático 12.7 muda como resultado de cada uma das seguintes etapas:
- Excluir 11.
 - Excluir 31.
 - Inserir 1.
 - Inserir 15.

12.9 Dê o pseudocódigo para a exclusão de entradas de uma estrutura de hash extensível, incluindo detalhes de quando e como unir os buckets. Não se preocupe em reduzir o tamanho da tabela de endereços de bucket.

- 12.10 Sugira um modo eficiente de testar se a tabela de endereços de bucket no hashing extensível pode ser reduzida em tamanho, armazenando uma contagem extra com a tabela de endereços de bucket. Dê os detalhes de como a contagem deve ser mantida quando os buckets são divididos, unidos ou excluídos.
(Nota: A redução do tamanho da tabela de endereços de bucket é uma operação dispendiosa, e as inserções subsequentes podem fazer com que a tabela cresça novamente. Portanto, é melhor não reduzir o tamanho assim que for possível, mas, em vez disso, reduzir apenas se o número de entradas de índice se tornar pequeno em comparação com o tamanho da tabela de endereços de bucket.)

12.11 Considere a relação *conta* mostrada na Figura 12.25.

- Construa um índice de mapa de bits sobre os atributos *nome_agência* e *saldo*, dividindo os valores de *saldo* em 4 intervalos: abaixo de 250, 250 até abaixo de 500, 500 até abaixo de 750 e 750 em diante.
- Considere uma consulta que solicita todas as contas em Downtown com um saldo superior ou igual a 500. Esboce as etapas para responder a consulta, e mostre os mapas de bit finais e intermediários construídos para responder a consulta.

- 12.12 Suponha que você tenha uma relação com n , tuplas em que uma árvore B* secundária deve ser construída.

- Dê uma fórmula para o custo de construção do índice de árvore B^* inserindo um registro de cada vez. Suponha que cada página manterá uma média de f entradas, e que todos os níveis da árvore acima da folha estejam na memória.
- Considerando um tempo de acesso ao disco de 10 milissegundos, qual é o custo da construção de índice em uma relação com 10 milhões de registros?
- Sugira um modo mais eficiente de construir o índice de baixo para cima, construindo primeiro o nível de folha inteiro e depois níveis mais altos, um por vez. Suponha que você tenha uma função que possa classificar, de modo eficiente, um conjunto muito grande de registros, mesmo o conjunto seja maior do que pode caber na memória. (Esses algoritmos de ordenação são descritos mais adiante, na seção "Classificação" do Capítulo 13 e, considerando uma quantidade razoável de memória principal, eles têm um custo de aproximadamente uma operação de E/S por bloco.)

Exercícios

- Quando é preferível usar um índice denso no lugar de um índice esparsão? Explique sua resposta.
- Qual é a diferença entre um índice agrupado e um índice secundário?
- Para cada árvore B^{**} do Exercício prático 12.3, mostre as etapas envolvidas nas seguintes consultas:
 - Encontrar registros com um valor de chave de busca igual a 11.
 - Encontrar registros com um valor de chave de busca entre 7 e 17, inclusive.
- A solução apresentada na seção "Chaves de busca não exclusivas" para lidar com chaves de busca não exclusivas acrescentou um atributo extra à chave de busca. Que efeito essa mudança tem sobre a altura da árvore B^* ?
- Explique a distinção entre o hashing fechado e aberto. Discuta os méritos relativos de cada técnica nas aplicações de banco de dados.
- Quais são os casos de estouro de bucket em uma organização de arquivo de hash? O que pode ser feito para reduzir a ocorrência de estouros de bucket?
- Por que uma estrutura de hash não é a melhor escolha para uma chave de busca em que as consultas de intervalo são prováveis?
- Suponha que haja uma relação $R(A,B,C)$, com um índice de árvore B^* com chave de busca (A,B) .

- Qual é o custo no pior caso para encontrar os registros que satisfazem $10 < A < 50$, usando esse índice, em termos do número de registros apanhados n_1 e a altura h da árvore?
- Qual é o custo no pior caso para encontrar os registros que satisfazem $10 < A < 50 \wedge 5 < B < 10$ usando esse índice, em termos do número de registros n_2 que satisfazem essa seleção, bem como n_1 e h , definidos anteriormente?
- Sob que condições sobre n_1 e n_2 o índice seria um modo eficiente de encontrar registros que satisfaçam $10 < A < 50 \wedge 5 < B < 10$?
- Suponha que você tenha de criar um índice de árvore B^* sobre uma grande quantidade de nomes, em que o tamanho máximo de um nome pode ser muito grande (digamos, 40 caracteres) e, na média, o nome seja grande (digamos 10 caracteres). Explique como a compactação do prefixo pode ser usada para melhorar o fanout médio dos nós internos.
- Por que os nós de folha de uma organização de arquivo de árvore B^* poderiam perder sua seqüencialidade? Sugira como a organização do arquivo pode ser reorganizada para restaurar a seqüencialidade?
- Suponha que uma relação seja armazenada em uma organização de arquivo de árvore B^* . Suponha que os índices secundários armazenasse identificadores de registro que são ponteiros para registros no disco.
 - Qual seria o efeito de uma divisão de página ocorresse na organização do arquivo?
 - Qual seria o custo de atualizar todos os registros afetados em um índice secundário?
 - Como o uso da chave de busca da organização de arquivo como um identificador de registro lógico soluciona esse problema?
 - Qual é o custo extra devido ao uso de tais identificadores de registro lógico?
- Mostre como calcular a existência de mapas de bits a partir de outros mapas de bits. Certifique-se de que sua técnica funciona mesmo na presença de valores nulos, usando um mapa de bits para o valor nulo.
- Como a criptografia de dados afeta os esquemas de índice? Em particular, como ela poderia afetar os esquemas que tentam armazenar dados em ordem classificada?
- Nossa descrição do hashing estático considera que uma grande faixa contígua de blocos de disco pode ser alocada a uma tabela de hash estática. Suponha que você possa alocar apenas C blocos contíguos. Sugira como implementar a tabela de hash, se ela puder ser maior do que C blocos. O acesso a um bloco ainda deverá ser eficiente.

Notas bibliográficas

As discussões sobre as estruturas de dados básicas na indexação e no hashing podem ser encontradas em Cormen *et al.* [1990]. Os índices de árvore B foram introduzidos inicialmente em Bayer [1972] e Bayer e McCreight [1972]. As árvores B^{*} são discutidas em Comer [1979], Bayer e Unterauer [1977] e Knuth [1973]. As notas bibliográficas no Capítulo 16 oferecem referências à pesquisa sobre a permissão dos acessos concorrentes e atualizações em árvores B^{*}. Gray e Reuter [1993] oferecem uma boa descrição das questões na implementação de árvores B^{*}.

Várias estruturas alternativas de árvore e tipo árvore foram propostas. Tries são árvores cuja estrutura é baseada nos "dígitos" das chaves (por exemplo, um índice de marcador de dicionário, que tem uma entrada para cada letra). Essas árvores podem ser balanceadas no mesmo sentido das árvores B^{*}. Tries são discutidas por Ramesh *et al.* [1989], Orestein [1982], Litwin [1981] e Fredkin [1960]. O trabalho relacionado inclui as árvores B de Lomet [1981].

Knuth [1973] analisa uma grande quantidade de técnicas de hashing diferentes. Existem vários esquemas de has-

hing dinâmico. O hashing extensível foi introduzido por Fagin *et al.* [1979]. O hashing linear foi introduzido por Litwin [1978, 1980]. Uma comparação de desempenho com hashing extensível é fornecida por Rathi *et al.* [1990]. Uma alternativa dada por Ramakrishna e Larson [1989] permite a recuperação em um único acesso ao disco, ao preço de uma sobrecarga alta para uma pequena fração de modificações do banco de dados. O hashing particionado é uma extensão do hashing para vários atributos, e é explicado em Rivest [1976], Byrgard [1976] e Burkhard [1979].

Vitter [2001] oferece um estudo abrangente sobre as estruturas e os algoritmos de dados na memória externa.

Os índices de mapa de bits, e suas variantes chamadas índice de bits e índices de projeção, são descritos em O'Neil e Quass [1997]. Eles foram introduzidos inicialmente no gerenciador de arquivos IBM Model 204, na plataforma AS 400. Eles oferecem aumentos de velocidade muito grandes em certos tipos de consultas, e hoje são implementados na maioria dos sistemas de banco de dados. A pesquisa recente sobre os índices de mapa de bits inclui Wu e Buchmann [1998], Chan e Ioannidis [1998], Chan e Ioannidis [1999] e Johnson [1999].