Snow666

Elmasri • Navathe

Sistemas de banco de dados

6ª Edição

Ramez Elmasri
Departamento de Ciência da Computação e Engenharia
Universidade do Texas em Arlington
Shamkant B. Navathe
Faculdade de Computação
Georgia Institute of Technology

Tradução:
Daniel Vieira
Revisão técnica:
Enzo Seraphim
Thatyana de Faria Piola Seraphim

Professores Doutores do Instituto de Engenharia de Sistemas e Tecnologias da Informação — Universidade Federal de Itajubá

PEARSON

São Paulo

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

ABDR
COPIA NÃO AUTORIZADA É CRIME
RESPEITE O DIREITO AUTORAL
EDITORA AFILIADA

Snow666

capítulo 17

Armazenamento de disco, estruturas de arquivo básicas e hashing

Os bancos de dados são armazenados fisicamente como arquivos de registros, que em geral ficam em discos magnéticos. Este capítulo e o seguinte tratam da organização dos bancos de dados em locais de armazenamento e as técnicas para acessá-los de modo eficiente usando diversos algoritmos, alguns dos quais exigindo estruturas de dados auxiliares, chamadas *índices*. Essas estruturas costumam ser conhecidas como **estruturas físicas de arquivo de banco de dados**, e estão no nível físico da arquitetura de três esquemas descrita no Capítulo 2. Começamos a Seção 17.1 introduzindo os conceitos de hierarquias de armazenamento de computador e como elas são usadas nos sistemas de banco de dados. A Seção 17.2 é dedicada a uma descrição dos dispositivos de armazenamento de disco magnético e suas características, e também descrevemos rapidamente os dispositivos de armazenamento de fita magnética. Depois de discutir diferentes tecnologias de armazenamento, voltamos nossa atenção para os métodos para organizar fisicamente os dados nos discos. A Seção 17.3 aborda a técnica de buffering duplo, que é usada para agilizar a recuperação de múltiplos blocos de disco. Na Seção 17.4, discutimos diversas maneiras de formatar e armazenar registros de arquivo no disco. A Seção 17.5 discute os diversos tipos de operações que são normalmente aplicadas aos registros do arquivo. Apresentamos três métodos principais para organizar registros de arquivo no disco: registros desordenados, na Seção 17.6; registros ordenados, na Seção 17.7; e registros com hashing, na Seção 17.8.

A Seção 17.9 apresenta rapidamente os arquivos de registros mistos e outros métodos principais para organizar registros, como as B-trees. Estas são particularmente relevantes para o armazenamento de bancos de dados orientados a objeto, que discutimos no Capítulo 11. A Seção 17.10 descreve o RAID (*Redundant Arrays of Inexpensive (ou Independent) Disks*) — uma arquitetura de sistema de armazenamento de dados que normalmente é usada em grandes organizações para obter melhor confiabilidade e desempenho. Por fim, na Seção 17.11, descrevemos três desenvolvimentos na área de sistemas de armazenamento: **área de armazenamento em rede** (SAN, do inglês, **Storage Area Networks**), armazenamento conectado à rede (NAS, do inglês, **Network-Attached Storage**) e iSCSI (**Internet SCSI — Small Computer System Interface**), a tecnologia mais recente, que torna as redes de área de armazenamento mais acessíveis sem o uso da infraestrutura canais de fibra e, portanto, está obtendo grande aceitação na indústria. No final do capítulo há um resumo. No Capítulo 18, discutimos as técnicas para criar estruturas de dados auxiliares, chamadas de índices, que agilizam a busca e a recuperação de registros. Essas técnicas envolvem o armazenamento de dados auxiliares, chamados arquivos de índice, além dos próprios registros do arquivo.

Os capítulos 17 e 18 podem ser folheados ou mesmo omitidos pelos leitores que já estudaram organizações e indexação de arquivos em outro curso. O material abordado aqui, em particular as seções 17.1 a 17.8, é necessário para se entender os capítulos 19 e 20, que tratam do processamento, da otimização de consulta e do ajuste do banco de dados para melhorar o desempenho das consultas.

17.1 Introdução

A coleção de dados que compõem um banco de dados computadorizado deve ser armazenada fisicamente em algum **meio de armazenamento** no computador. O software de SGBD pode então recuperar, atualizar e processar esses dados conforme a necessidade. A mídia de armazenamento de computador

Snow666

forma uma *hierarquia de armazenamento* que inclui duas categorias principais:

- **Armazenamento primário.** Essa categoria inclui a mídia de armazenamento que pode ser operada diretamente pela *unidade central de processamento* (CPU) do computador, como a memória principal do computador e memórias cache menores, porém mais rápidas. O armazenamento primário normalmente oferece acesso rápido aos dados, mas tem capacidade de armazenamento limitada. Embora as capacidades da memória principal estejam crescendo rapidamente nos últimos anos, elas ainda são mais caras e têm menos capacidade de armazenamento do que os dispositivos de armazenamento secundários e terciários.
- **Armazenamento secundário e terciário.** Essa categoria inclui discos magnéticos, discos ópticos (CD-ROMs, DVDs e outros meios de armazenamento semelhantes) e fitas. As unidades de disco rígido são classificadas como armazenamento secundário, enquanto a mídia removível, como discos ópticos e as fitas, é considerada armazenamento terciário. Esses dispositivos costumam ter uma capacidade maior, menor custo e oferecem acesso mais lento aos dados do que os dispositivos de armazenamento primários. Os dados no armazenamento secundário ou terciário não podem ser processados diretamente pela CPU; primeiro, eles precisam ser copiados para o armazenamento primário e, depois, processados pela CPU.

Primeiro, damos uma visão geral dos diversos dispositivos de armazenamento usados para armazenamento primário e secundário na Seção 17.1.1 e, depois, discutimos como os bancos de dados normalmente são tratados na hierarquia de armazenamento na Seção 17.1.2.

17.1.1 Hierarquias de memória e dispositivos de armazenamento

Em um sistema de computador moderno, os dados residem e são transportados por uma hierarquia de meios de armazenamento. A memória de velocidade mais alta é a mais cara e, portanto, está disponível com a menor capacidade. A memória de velocidade mais lenta é o armazenamento em fita off-line, que basicamente está disponível em capacidade de armazenamento indefinida.

No *nível de armazenamento primário*, a hierarquia de memória inclui, no extremo mais caro, a **memória cache**, que é uma RAM (*Random Access Memory*) estática. A memória cache normalmente é usada pela CPU para agilizar a execução de instruções de programa usando técnicas como pré-busca e *pipelining*. O próximo nível de armazenamento primário é a DRAM (*Dynamic* RAM), que oferece a área de trabalho principal para a CPU, para manter instruções de programa e dados. Ela é popularmente chamada de **memória principal**. A vantagem da DRAM é seu baixo custo, que continua a diminuir; a desvantagem é sua volatilidade[1] e menor velocidade em comparação com a RAM estática. No *nível de armazenamento secundário e terciário*, a hierarquia inclui discos magnéticos, bem como **armazenamento em massa** na forma de dispositivos de CD-ROM (*Compact Disk–Read-Only Memory*) e DVD (*Digital Video Disk* ou *Digital Versatile Disk*) e, por fim, fitas no extremo mais barato da hierarquia. A **capacidade de armazenamento** é medida em kilobytes (Kbyte ou 1.000 bytes), megabytes (MB ou 1 milhão de bytes), gigabytes (GB ou 1 bilhão de bytes) e até mesmo terabytes (1.000 GB). A palavra petabyte (1.000 terabytes ou 10**15 bytes) agora está se tornando relevante no contexto de repositórios muito grandes de dados na física, astronomia, ciências terrestres e outras aplicações científicas.

Os programas residem e são executados na DRAM. Em geral, grandes bancos de dados permanentes residem no armazenamento secundário (discos magnéticos) e partes do banco de dados são lidas e escritas de buffers na memória principal conforme a necessidade. Atualmente, os computadores pessoais e as estações de trabalho possuem grandes memórias principais de centenas de megabytes de RAM ou DRAM, de modo que está se tornando possível carregar uma grande parte do banco de dados na memória principal. Oito a 16 GB de memória principal em um único servidor está se tornando algo comum. Em alguns casos, bancos de dados inteiros podem ser mantidos na memória principal (com uma cópia de backup no disco magnético), levando a **bancos de dados de memória principal**. Estes são particularmente úteis em aplicações de tempo real que exigem tempos de resposta extremamente rápidos. Um exemplo são as aplicações de comutação de telefone, que armazenam bancos de dados que contêm informações de roteamento e linha na memória principal.

Entre a DRAM e o armazenamento em disco magnético, outra forma de memória, a **memória flash**, está se tornando comum, principalmente porque ela é não volátil. As memórias flash são de alta

[1] A memória volátil normalmente perde seu conteúdo em caso de falta de energia, mas com a memória não volátil isso não acontece.

Snow666

densidade, alto desempenho, e usam a tecnologia EEPROM (*Electrically Erasable Programmable Read-Only Memory*). A vantagem da memória flash é a velocidade de acesso rápida; a desvantagem é que um bloco inteiro precisa ser apagado e gravado simultaneamente. As placas de memória flash estão aparecendo como o meio de armazenamento de dados em aparelhos domésticos com capacidades variando de alguns megabytes até alguns gigabytes. Estão presentes em câmeras, MP3 players, telefones celulares, PDAs, e assim por diante. Unidades flash USB (*Universal Serial Bus*) se tornaram o meio mais portátil para transportar dados entre computadores pessoais; elas têm um dispositivo de armazenamento de memória flash integrado a uma interface USB.

Discos de CD-ROM armazenam dados opticamente e são lidos por um laser. Os CD-ROMs contêm dados pré-gravados que não podem ser modificados. Os discos WORM (*Write-Once-Read-Many*) são uma forma de armazenamento óptico usado para arquivar dados; eles permitem que os dados sejam gravados uma vez e lidos qualquer número de vezes sem a possibilidade de apagamento. Eles mantêm cerca de meio gigabyte de dados por disco e duram muito mais do que os discos magnéticos.[2] **Memórias de jukebox óptico** utilizam um conjunto de placas de CD-ROM, que são carregadas em unidades por demanda. Embora os jukeboxes ópticos tenham capacidades na ordem de centenas de gigabytes, seus tempos de recuperação estão na ordem de centenas de milissegundos, muito mais lento do que os discos magnéticos. Esse tipo de armazenamento continua a diminuir devido à rápida diminuição no custo e ao aumento nas capacidades dos discos magnéticos. O DVD é outro padrão para discos ópticos, permitindo 4,5 a 15 GB de armazenamento por disco. A maioria das unidades de disco de computador pessoal agora lê discos de CD-ROM e DVD. Normalmente, as unidades são CD-R (*Compact Disk Recordable*) que podem criar CD-ROMs e CDs de áudio (*Compact Disks*), bem como gravar DVDs.

Finalmente, as **fitas magnéticas** são usadas para arquivamento e armazenamento de backup dos dados. Os **jukeboxes de fita** — que contêm um banco de fitas que são catalogadas e podem ser carregadas automaticamente nas unidades de fita — estão se tornando populares como **armazenamento terciário**, para manter terabytes de dados. Por exemplo, o sistema satélite de observação da Terra (**EOS — Earth Observation Satellite**) da NASA armazena bancos de dados arquivados dessa forma.

Muitas organizações grandes já estão achando normal ter bancos de dados com tamanhos na ordem dos terabytes. O termo **banco de dados muito grande** não pode mais ser definido com exatidão, pois as capacidades de armazenamento em disco estão aumentando e os custos, diminuindo. Logo, o termo pode ser reservado para bancos de dados com dezenas de terabytes.

17.1.2 Armazenamento de bancos de dados

Os bancos de dados costumam armazenar grande quantidade de dados que precisam persistir por longos períodos de tempo e, portanto, eles costumam ser considerados **dados persistentes**. Partes desses dados são acessadas e processadas repetidamente durante esse período. Isso é contrário à noção de **dados transientes**, que persistem apenas por um tempo limitado durante a execução do programa. A maioria dos bancos de dados é armazenada de maneira permanente (ou *persistentemente*) no armazenamento secundário do disco magnético, pelos seguintes motivos:

- Em geral, os bancos de dados são muito grandes para caber inteiramente na memória principal.
- As circunstâncias que causam perda permanente de dados armazenados surgem com menos frequência para o armazenamento de disco secundário do que para o armazenamento primário. Logo, referimo-nos ao disco — e a outros dispositivos de armazenamento secundário — como **armazenamento não volátil**, enquanto a memória principal normalmente é chamada de **armazenamento volátil**.
- O custo de armazenamento por unidade de dados está na ordem de grandeza menor para o armazenamento de disco secundário do que para o armazenamento primário.

Algumas das tecnologias mais novas — como discos ópticos, DVDs e jukeboxes de fita — provavelmente oferecem alternativas viáveis ao uso de discos magnéticos. No futuro, portanto, os bancos de dados poderão residir em diferentes níveis de hierarquia de memória, com base naquelas descritas na Seção 17.1.1. Contudo, já se sabe que os discos magnéticos continuarão a ser o meio de escolha principal para bancos de dados grandes por muitos anos. Logo, é importante estudar e entender as propriedades e as características dos discos magnéticos e o modo como os arquivos podem ser organizados neles a fim de projetar bancos de dados eficazes, com desempenho aceitável.

[2] Suas velocidades de rotação são mais baixas (em torno de 400 rpm), gerando maiores atrasos de latência e baixas taxas de transferência (em torno de 100 a 200 KB/segundo).

Sn❄w66

As fitas magnéticas são frequentemente usadas como meio de armazenamento para o backup de bancos de dados, pois o armazenamento em fita custa ainda menos que o armazenamento em disco. No entanto, o acesso aos dados na fita é muito lento. Os dados armazenados nas fitas são **off-line**; ou seja, alguma intervenção por um operador — ou um dispositivo de carga automática — para carregar uma fita é necessário antes que os dados se tornem disponíveis. Ao contrário, os discos são dispositivos **on-line**, que podem ser acessados diretamente a qualquer momento.

As técnicas utilizadas para armazenar grande quantidade de dados estruturados em disco são importantes para os projetistas de banco de dados, para o DBA e para implementadores de um SGBD. Os projetistas de banco de dados e o DBA precisam conhecer as vantagens e desvantagens de cada técnica de armazenamento quando projetam, implementam e operam um banco de dados em um SGBD específico. Em geral, o SGBD tem várias opções disponíveis para organizar os dados. O processo de **projeto físico de banco de dados** envolve a escolha das técnicas de organização de dados em particular que mais se ajustam a determinados requisitos da aplicação dentre as opções. Os implementadores de sistema de SGBD devem estudar as técnicas de organização de dados de modo que possam implementá-las de modo eficaz e, portanto, oferecer ao DBA e aos usuários do SGBD opções suficientes.

As aplicações de banco de dados típicas só precisam de uma pequena parte do banco de dados de cada vez para processamento. Sempre que certa parte dos dados é necessária, ela precisa ser localizada no disco, copiada para a memória principal para processamento e, depois, reescrita para o disco se os dados forem alterados. Os dados armazenados no disco são organizados como **arquivos de registros**. Cada registro é uma coleção de valores de dados que podem ser interpretados como fatos sobre entidades, seus atributos e relacionamentos. Os registros devem ser armazenados em disco de uma maneira que torne possível localizá-los de modo eficiente quando necessário.

Existem várias **organizações de arquivo primário**, que determinam como os registros de arquivo são *colocados fisicamente* no disco, *e daí como os registros podem ser acessados*. Um *arquivo de heap* (ou *arquivo desordenado*) coloca os registros no disco sem qualquer ordem em particular, acrescentando novos registros ao final do arquivo, enquanto um *arquivo classificado* (ou *arquivo sequencial*) mantém os registros ordenados pelo valor de um campo em particular (chamado *campo de classificação*). Um *arquivo em hashing* usa uma função de hash aplicada a um campo em particular (chamado *chave hash*) para determinar o posicionamento de um registro no disco. Outras organizações de arquivo primárias, como *B-trees*, usam estruturas em árvore. Discutimos as principais organizações de arquivo nas seções 17.6 até 17.9. Uma **organização secundária** ou **estrutura de acesso auxiliar** permite acesso eficiente aos registros do arquivo com base em *campos alternativos*, além dos que foram usados para a organização de arquivo primário. A maioria destes existe como índices e será discutida no Capítulo 18.

17.2 Dispositivos de armazenamento secundários

Nesta seção, descrevemos algumas características dos dispositivos de armazenamento em disco magnético e fita magnética. Os leitores que já estudaram esses dispositivos podem simplesmente folhear esta seção.

17.2.1 Descrição de hardware dos dispositivos de disco

Os discos magnéticos são usados para armazenar grande quantidade de dados. A unidade de dados mais básica no disco é um único **bit** de informação. Ao magnetizar uma área do disco de certas maneiras, pode-se fazer que ele represente um valor de bit de 0 (zero) ou 1 (um). Para codificar a informação, os bits são agrupados em **bytes** (ou **caracteres**). Os tamanhos de byte normalmente variam de 4 a 8 bits, dependendo do computador e do dispositivo. Consideramos que um caractere é armazenado em um único byte, e usamos os termos *byte* e *caractere* para indicar a mesma coisa. A **capacidade** de um disco é o número de bytes que ele pode armazenar, que em geral é muito grande. Pequenos disquetes usados com microcomputadores costumam manter de 400 KB a 1,5 MB; mas eles estão rapidamente saindo de circulação. Os discos rígidos para computadores pessoais normalmente mantêm de várias centenas de MB até dezenas de GB; e grandes pacotes de disco usados com servidores e mainframes têm capacidades de centenas de GB. As capacidades de disco continuam a crescer à medida que a tecnologia se aperfeiçoa.

Independentemente de sua capacidade, todos os discos são feitos de um material magnético modelado como um disco circular fino, como mostra a Figura 17.1(a), e protegidos por uma camada de plástico ou acrílico.

Snow666

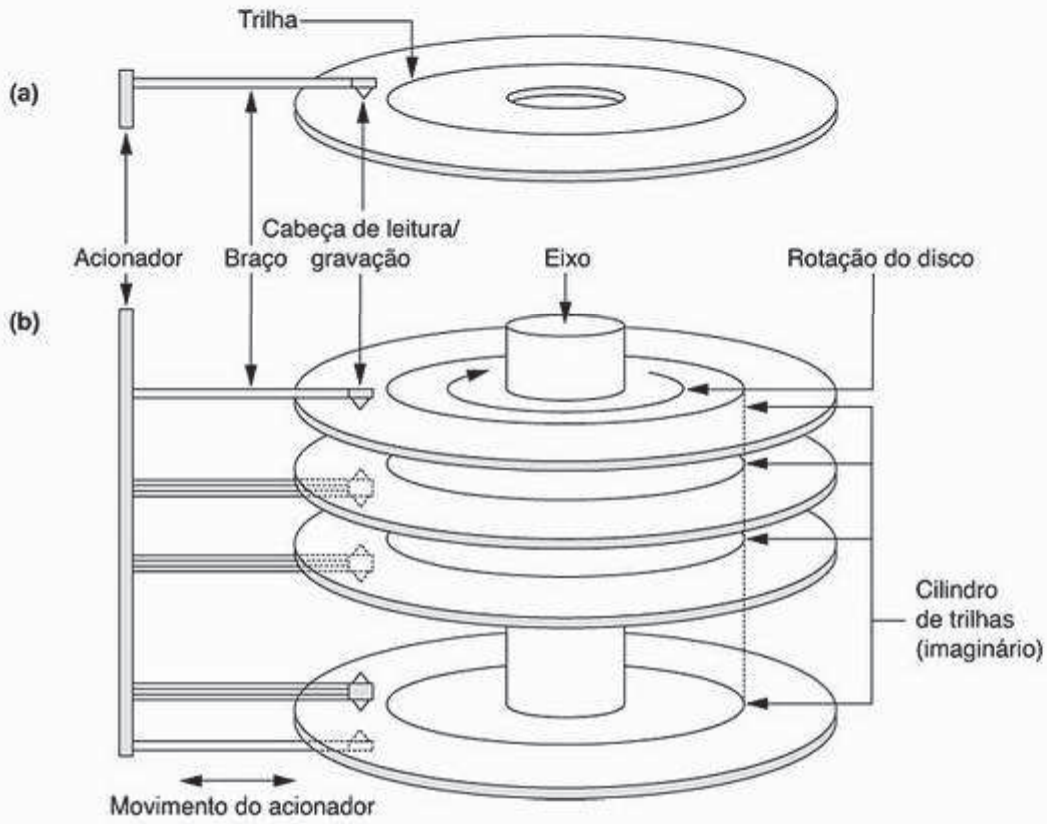

Figura 17.1
(a) Um disco de face simples com hardware de leitura/gravação. (b) Um disk pack com hardware de leitura/gravação.

Um disco é de **face simples** se armazenar informações apenas em uma de suas superfícies, e de **face dupla** se as duas superfícies forem usadas. Para aumentar a capacidade de armazenamento, os discos são montados em um **disk pack**, como mostra a Figura 17.1(b), que pode incluir muitos discos e, portanto, muitas superfícies. As informações são armazenadas em uma superfície do disco em círculos concêntricos de *pequena largura*,[3] cada um com um diâmetro distinto. Cada círculo é chamado de **trilha**. Em disk packs, as trilhas com o mesmo diâmetro nas diversas superfícies formam um **cilindro**, devido à forma que elas teriam se fossem conectadas no espaço. O conceito de cilindro é importante porque os dados armazenados em um cilindro podem ser recuperados muito mais rapidamente do que se fossem distribuídos entre diferentes cilindros.

O número de trilhas em um disco varia de algumas centenas até alguns milhares, e a capacidade de cada trilha normalmente varia de dezenas de KB até 150 KB. Como uma trilha em geral contém uma grande quantidade de informações, ela é dividida em blocos ou setores menores. A divisão de uma trilha em **setores** é fixada na superfície do disco e não pode ser alterada. Um tipo de organização de setor, como mostra a Figura 17.2(a), chama uma parte da trilha que se estende por um ângulo fixo no centro de um setor. Várias outras organizações de setor são possíveis, e uma delas é fazer que os setores se estendam por ângulos menores no centro à medida que se move para fora, mantendo assim uma densidade de gravação uniforme, como mostra a Figura 17.2(b). Uma técnica chamada ZBR (*Zone Bit Recording*) permite que um intervalo de cilindros tenha o mesmo número de setores por arco. Por exemplo, os cilindros 0-99 podem ter um setor por trilha, 100-199 podem ter dois por trilha, e assim por diante. Nem todos os discos têm suas trilhas divididas em setores.

A divisão de uma trilha em **blocos de disco** (ou **páginas**) de mesmo tamanho é definida pelo sistema operacional durante a **formatação** (ou **inicialização**) do disco. O tamanho do bloco é fixado no decorrer da inicialização e não pode ser trocado dinamicamente. Os tamanhos de bloco de disco típicos variam de

[3] Em alguns discos, os círculos agora são conectados em um tipo de espiral contínua.

Snow666

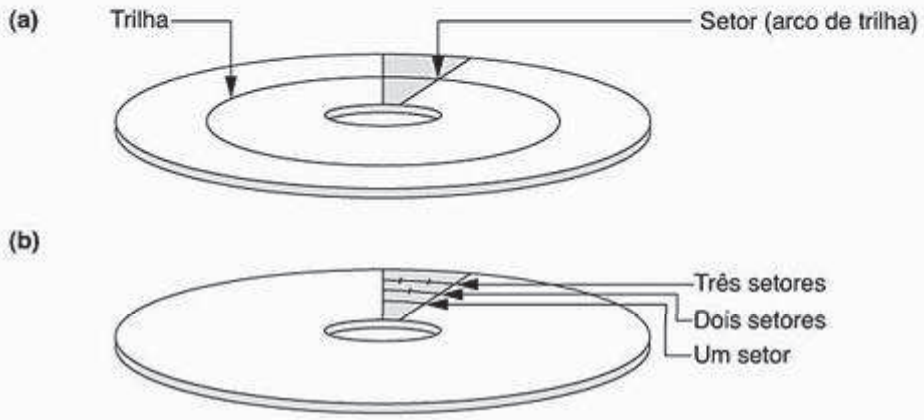

Figura 17.2
Diferentes organizações de setor no disco. (a) Setores se estendendo por um ângulo fixo. (b) Setores mantendo uma densidade de gravação uniforme.

512 a 8.192 bytes. Um disco com setores fixos costuma ter os setores subdivididos em blocos durante a inicialização. Os blocos são separados por **lacunas entre blocos** de tamanho fixo, que incluem informações de controle especialmente codificadas, gravadas durante a inicialização do disco. Essa informação é usada para determinar qual bloco da trilha segue cada lacuna entre blocos. A Tabela 17.1 ilustra as especificações dos discos típicos usados em grandes servidores na indústria. Os prefixos 10K e 15K nos nomes de disco referem-se às velocidades de rotação em rpm (rotações por minuto).

Tabela 17.1
Especificações de discos Cheetah típicos de alto nível da Seagate.

Descrição	Cheetah 15K.6	Cheetah NS 10K
Número do modelo	ST3450656SS/FC	ST3400755FC
Altura	25,4 mm	26,11 mm
Largura	101,6 mm	101,85 mm
Comprimento	146,05 mm	147 mm
Peso	0,709 kg	0,771 kg
Capacidade		
Capacidade formatada	450 GB	400 GB
Configuração		
Número de discos (físicos)	4	4
Número de cabeças (físicas)	8	8
Desempenho		
Taxas de transferência		
Taxa de transferência interna (mínima)	1.051 Mb/s	
Taxa de transferência interna (máxima)	2.225 Mb/s	1.211 Mb/s
Tempo médio entre falhas (MTBF)		1,4 M horas
Tempos de busca		
Tempo de busca médio (leitura)	3,4 ms (típica)	3,9 ms (típica)
Tempo de busca médio (gravação)	3,9 ms (típica)	4,2 ms (típica)
Busca trilha a trilha, leitura	0,2 ms (típica)	0,35 ms (típica)
Busca trilha a trilha, gravação	0,4 ms (típica)	0,35 ms (típica)
Latência média	2 ms	2,98 msec

Fonte: Cortesia da Seagate Technology

Snow666

Há uma melhoria contínua na capacidade de armazenamento e taxas de transferência associadas aos discos. Eles também estão ficando cada vez mais baratos — hoje, custam apenas uma fração de um dólar por megabyte de armazenamento em disco. Os custos estão reduzindo tão rapidamente que já chegaram a 0,025 centavos/MB — que se traduz em US$ 0,25/GB e US$ 250/TB.

Um disco é um dispositivo endereçável por *acesso aleatório*. A transferência de dados entre a memória principal e o disco ocorre em unidades de blocos de disco. O **endereço de hardware** de um bloco — uma combinação de número de cilindro, número de trilha (número de superfície no cilindro em que a trilha está localizada) e número de bloco (dentro da trilha) é fornecido ao hardware de E/S (entrada/saída) do disco. Em muitas unidades de disco modernas, um único número, chamado LBA (*Logical Block Address*), que é um número entre 0 e n (considerando que a capacidade total do disco é $n + 1$ blocos), é mapeado automaticamente para o bloco correto pelo controlador da unidade de disco. O endereço de um **buffer** — uma área reservada contígua no armazenamento principal, que mantém um bloco de disco — também é fornecido. Para um comando de **leitura**, o bloco de disco é copiado para o buffer, ao passo que, para um comando de **gravação**, o conteúdo do buffer é copiado para o bloco de disco. Às vezes, vários blocos contíguos podem ser transferidos como uma unidade, denominada **cluster**. Nesse caso, o tamanho do buffer é ajustado para corresponder ao número de bytes no cluster.

O mecanismo de hardware real que lê ou grava um bloco é a **cabeça de leitura/gravação** do disco, que faz parte de um sistema chamado **unidade de disco**. Um disco ou disk pack é montado na unidade de disco, que inclui um motor que gira os discos. Uma cabeça de leitura/gravação inclui um componente eletrônico conectado a um **braço mecânico**. Os disk packs com superfícies múltiplas são controlados por várias cabeças de leitura/gravação — uma para cada superfície, como mostra a Figura 17.1(b). Todos os braços são conectados a um **acionador** conectado a outro motor elétrico, que move as cabeças de leitura/gravação em harmonia e as posiciona exatamente sobre o cilindro de trilhas especificado em um endereço de bloco.

As unidades de disco rígido giram o disk pack continuamente a uma velocidade constante (em geral, variando entre 5.400 e 15.000 rpm). Quando a cabeça de leitura/gravação está posicionada na trilha correta e o bloco especificado no endereço de bloco move-se sob a cabeça de leitura/gravação, o componente eletrônico da cabeça de leitura/gravação é ativado para transferir os dados. Algumas unidades de disco possuem cabeças de leitura/gravação fixas, com o número de cabeças correspondente ao de trilhas. Estes são chamados **discos de cabeça fixa**, enquanto as unidades de disco com um acionador são chamadas de **discos de cabeça móvel**. Para os discos de cabeça fixa, uma trilha ou cilindro é selecionado eletronicamente, passando para a cabeça de leitura/gravação apropriada, em vez de por um movimento mecânico; em consequência, isso é muito mais rápido. Porém, o custo das cabeças de leitura/gravação adicionais é muito alto, de modo que os discos com cabeça fixa não são muito utilizados.

Um **controlador de disco**, comumente embutido na unidade de disco, controla a unidade de disco e a interliga ao sistema de computação. Uma das interfaces-padrão usadas hoje para unidades de disco nos PCs e estações de trabalho se chama **SCSI** (*Small Computer System Interface*). O controlador aceita comandos de E/S de alto nível e age de maneira apropriada para posicionar o braço e fazer que aconteça a ação de leitura/gravação. Para transferir um bloco de disco, dado seu endereço, o controlador de disco primeiro deve posicionar mecanicamente a cabeça de leitura/gravação na trilha correta. O tempo exigido para fazer isso é chamado **tempo de busca**. Os tempos de busca típicos são de 5 a 10 ms em desktops e de 3 a 8 ms em servidores. Depois disso, existe outro atraso — chamado de **atraso rotacional** ou **latência** — enquanto o início do bloco desejado gira até a posição sob a cabeça de leitura/gravação. Isso depende das rpm do disco. Por exemplo, a 15.000 rpm, o tempo por rotação é de 4 ms e o atraso rotacional médio é o tempo por meia rotação, ou 2 ms. A 10.000 rpm, o atraso rotacional médio aumenta para 3 ms. Finalmente, algum tempo adicional é necessário para transferir os dados; este é chamado **tempo de transferência de bloco**. Logo, o tempo total necessário para localizar e transferir um bloco qualquer, dado seu endereço, é a soma do tempo de busca, do atraso rotacional e do tempo de transferência de bloco. O tempo de busca e o atraso rotacional costumam ser muito maiores do que o tempo de transferência de bloco. Para tornar a transferência de múltiplos blocos mais eficiente, é comum transferir vários blocos consecutivos na mesma trilha ou cilindro. Isso elimina o tempo de busca e o atraso rotacional para todos, menos o primeiro bloco, e pode resultar em uma economia de tempo substancial quando vários blocos contíguos são transferidos. Normalmente, o fabricante do disco oferece uma **taxa de transferência em massa** a fim de calcular o tempo exigido para transferir blocos consecutivos. O Apêndice B contém uma discussão sobre esses e outros parâmetros de disco.

Snow666

O tempo necessário para localizar e transferir um bloco de disco está na ordem de milissegundos, em geral variando de 9 a 60 ms. Para blocos contíguos, localizar o primeiro bloco leva de 9 a 60 ms, mas transferir os blocos subsequentes pode levar apenas 0,4 a 2 ms cada. Muitas técnicas de busca tiram proveito da recuperação consecutiva de blocos ao procurar dados no disco. De qualquer forma, um tempo de transferência na ordem de milissegundos é considerado bastante alto em comparação com o tempo exigido para processar os dados na memória principal pelas CPUs atuais. Logo, a localização dos dados no disco é um *gargalo principal* nas aplicações de banco de dados. As estruturas de arquivo que discutimos aqui e no Capítulo 18 tentam *minimizar o número de transferências de bloco* necessárias para localizar e transferir os dados exigidos do disco para a memória principal. Colocar 'informações relacionadas' em blocos contíguos é o objetivo básico de qualquer organização de armazenamento no disco.

17.2.2 Dispositivos de armazenamento de fita magnética

Discos são dispositivos de armazenamento secundário de **acesso aleatório**, pois um bloco de disco qualquer pode ser acessado *aleatoriamente* depois que especificamos seu endereço. As fitas magnéticas são dispositivos de acesso sequencial; para acessar o enésimo bloco na fita, primeiro temos de varrer os $n - 1$ blocos anteriores. Os dados são armazenados em bobinas de fita magnética de alta capacidade, semelhantes às fitas de áudio ou vídeo. Uma unidade de fita precisa ler os dados ou gravá-los em uma **bobina de fita**. Em geral, cada grupo de bits que forma um byte é armazenado na fita, e os próprios bytes são armazenados consecutivamente nela.

Uma cabeça de leitura/gravação é usada para ler ou gravar dados na fita. Os registros de dados na fita também são armazenados em blocos — embora os blocos possam ser substancialmente maiores do que os dos discos, e as lacunas entre blocos também sejam muito grandes. Com densidades de fita típicas de 1.600 a 6.250 bytes por polegada, uma lacuna entre blocos típica[4] de 0,6 polegada corresponde a 960 até 3.750 bytes de espaço de armazenamento desperdiçado. É comum agrupar muitos registros em um bloco para melhorar a utilização do espaço.

A principal característica de uma fita é seu requisito de que acessemos os blocos de dados em **ordem sequencial**. Para chegar até um bloco no meio da bobina de fita, a fita é montada e depois varrida até que o bloco solicitado passe sob a cabeça de leitura/gravação. Por esse motivo, o acesso da fita pode ser lento e elas não são usadas para armazenar dados on-line, exceto para algumas aplicações especializadas. Porém, as fitas têm uma função muito importante — **backup** do banco de dados. Uma razão para haver backup é para manter cópias de arquivos de disco no caso de perda de dados por uma falha no disco, que pode acontecer se a sua cabeça de leitura/gravação tocar na superfície por defeito mecânico. Por esse motivo, os arquivos de disco são copiados periodicamente para a fita. Para muitas aplicações críticas on-line, como sistemas de reserva aérea, para evitar qualquer tempo de paralisação, são usados sistemas espelhados para manter três conjuntos de discos idênticos — dois em operação on-line e um como backup. Aqui, os discos off-line tornam-se um dispositivo de backup. Os três são usados de modo que possam ser trocados caso haja uma falha em uma das unidades de disco ativas. As fitas também podem ser utilizadas para armazenar arquivos de banco de dados excessivamente grandes. Os arquivos do banco de dados que raramente são usados ou estão desatualizados, mas são necessários para manutenção de registro histórico, podem ser **arquivados** em fita. Originalmente, as unidades de fita com bobina de meia polegada eram usadas para o armazenamento de dados, empregando as chamadas fitas de nove trilhas. Mais tarde, fitas magnéticas menores, de 8mm (semelhantes às usadas em filmadoras portáteis), que podem armazenar até 50 GB, bem como os cartuchos de dados de varredura helicoidal de 4mm e CDs e DVDs graváveis, tornaram-se mídia popular para o backup de arquivos de dados dos PCs e estações de trabalho. Eles também são usados para armazenar imagens e bibliotecas de sistemas.

O backup de bancos de dados corporativos, de modo que nenhuma informação de transação seja perdida, é uma tarefa importante. Atualmente, as bibliotecas de fitas com slots para várias centenas de cartuchos são usadas com Digital e Superdigital Linear Tapes (DLTs e SDLTs), com capacidades na ordem de centenas de gigabytes, que registram dados em trilhas lineares. Braços robóticos são usados para gravar em vários cartuchos em paralelo, utilizando várias unidades de fita com software de rotulagem automático para identificar os cartuchos de backup. Um exemplo de uma biblioteca gigante é o modelo SL8500 da Sun Storage Technology, que pode armazenar até 70 petabytes (1 petabyte = 1.000 TB) de dados usando até 448 unidades com uma taxa de vazão máxima de 193,2 TB/hora. Vamos adiar a discussão sobre a tecnologia de armazenamento de disco chamada RAID, e sobre as áreas de armazenamento em rede, armazenamento conectado à rede e sistemas de armazenamento iSCSI para o final do capítulo.

[4] Chamadas de *lacunas entre registros* em terminologia de fita.

Snow666

17.3 Buffering de blocos

Quando vários blocos precisam ser transferidos do disco para a memória principal e todos os endereços de bloco são conhecidos, vários buffers podem ser reservados na memória principal para agilizar a transferência. Enquanto um buffer está sendo lido ou gravado, a CPU pode processar dados em outro buffer, pois existe um processador (controlador) de E/S de disco separado que, uma vez iniciado, pode prosseguir para transferir um bloco de dados entre a memória e o disco independente e em paralelo com o processamento da CPU.

A Figura 17.3 ilustra como dois processos podem prosseguir em paralelo. Os processos A e B estão rodando **simultaneamente** em um padrão **intervalado**, onde os processos C e D estão rodando **simultaneamente** em um padrão **paralelo**. Quando uma única CPU controla vários processos, a execução paralela não é possível. Contudo, os processos ainda podem ser executados de maneira simultânea de uma forma intervalada. O buffering é mais útil quando os processos podem ser executados simultaneamente em um padrão paralelo, seja porque existe um processador de E/S de disco separado ou porque há vários processadores (CPUs).

A Figura 17.4 ilustra como a leitura e o processamento podem prosseguir em paralelo quando o tempo exigido para processar um bloco de disco na memória é menor que o tempo exigido para ler o próximo bloco e preencher um buffer. A CPU pode começar a processar um bloco quando sua transferência para a memória principal termina; ao mesmo tempo, o processador de E/S de disco pode estar lendo e transferindo o próximo bloco para um buffer diferente. Essa técnica é chamada de **buffering duplo**, e também pode ser usada para ler um fluxo contínuo de blocos do disco para a memória. O buffering duplo permite a leitura ou gravação contínua de dados em blocos de disco consecutivos, o que elimina o tempo de busca e o atraso rotacional para todas as transferências de bloco, com exceção da primeira. Além do mais, os dados ficam prontos para processamento, reduzindo assim o tempo de espera nos programas.

17.4 Gravando registros de arquivo no disco

Nesta seção, definimos os conceitos de registros, tipos de registro e arquivos. Depois, discutimos sobre as técnicas para gravar registros de arquivo no disco.

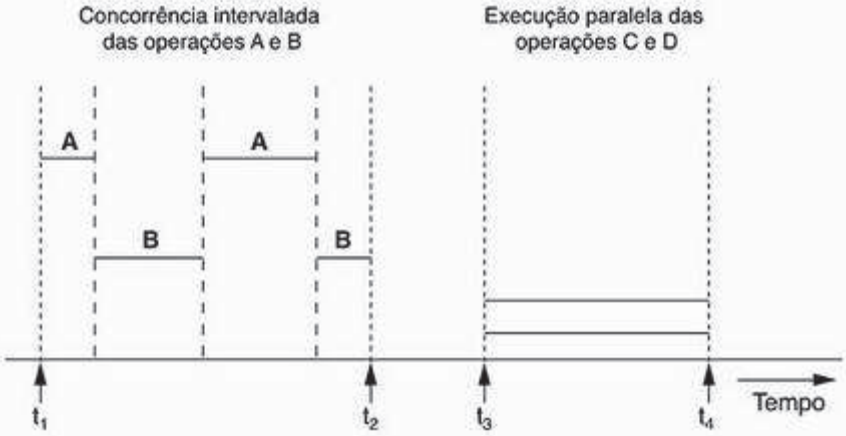

Figura 17.3
Concorrência intervalada *versus* execução paralela.

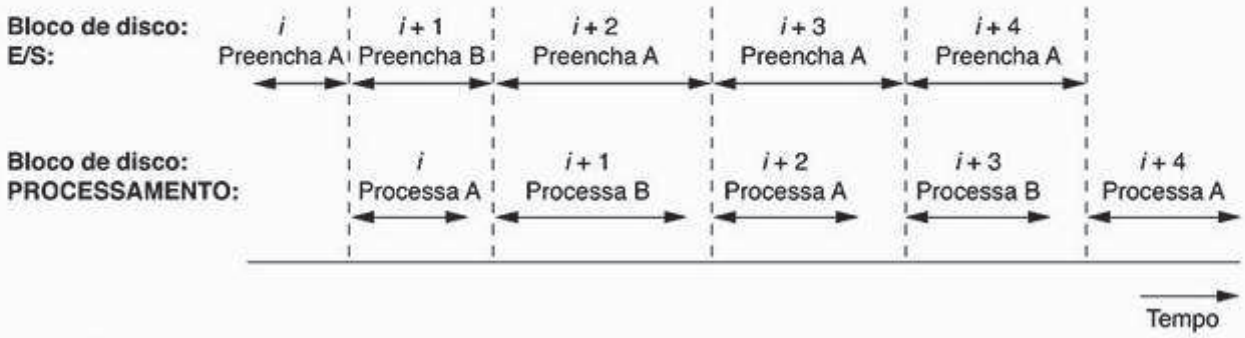

Figura 17.4
Uso de dois buffers, A e B, para a leitura do disco.

Snow666

17.4.1 Registros e tipos de registro

Os dados costumam ser armazenados na forma de **registros**. Cada registro contém uma coleção de **valores** ou **itens** de dados relacionados, no qual cada valor é formado por um ou mais bytes e corresponde a um **campo** em particular do registro. Os registros normalmente descrevem entidades e seus atributos. Por exemplo, um registro de FUNCIONARIO representa uma entidade de funcionário, e o valor de cada campo no registro especifica algum atributo desse funcionário, como Nome, Data_nascimento, Salario ou Supervisor. Uma coleção de nomes de campo e seus tipos de dados correspondentes constituem uma definição de **tipo de registro** ou **formato de registro**. Um **tipo de dado**, associado a cada campo, especifica os tipos de valores que um campo pode assumir.

O tipo de dado de um campo normalmente é um dos tipos de dado padrão usados na programação. Estes incluem os tipos de dados numéricos (inteiro, inteiro longo ou ponto flutuante), cadeia de caracteres (tamanho fixo ou variável), booleanos (tendo apenas valores 0 e 1, ou TRUE e FALSE) e, às vezes, tipos **data** e **tempo** especialmente codificados. O número de bytes exigidos para cada tipo de dado é fixo para determinado sistema de computação. Um inteiro pode exigir 4 bytes, um inteiro longo, 8 bytes, um número real, 4 bytes, um booleano, 1 byte, e uma data, 10 bytes (considerando um formato DD-MM-AAAA), e uma string de tamanho fixo de *k* caracteres, *k* bytes. As strings de tamanho variável podem exibir tantos bytes quantos caracteres existirem no valor de cada campo. Por exemplo, um tipo de registro FUNCIONARIO pode ser definido — usando a notação da linguagem de programação C — com a seguinte estrutura:

```
struct funcionario{
    char nome[30];
    char cpf[9];
    int salario;
    int cod_cargo;
    char departamento[20];
} ;
```

Em algumas aplicações de banco de dados, pode haver necessidade de armazenar itens de dados que consistem em grandes objetos não estruturados, que representam imagens, vídeo digitalizado ou streams de áudio, ou então texto livre. Estes são conhecidos como **BLOBs** (objetos binários grandes). Um item de dados BLOB costuma ser armazenado separadamente de seu registro, em um conjunto de blocos de disco, e um ponteiro para o BLOB é incluído no registro.

17.4.2 Arquivos, registros de tamanho fixo e registros de tamanho variável

Um **arquivo** é uma *sequência* de registros. Em muitos casos, todos os registros em um arquivo são do mesmo tipo de registro. Se cada registro no arquivo tem exatamente o mesmo tamanho (em bytes), o arquivo é considerado composto de **registros de tamanho fixo**. Se diferentes registros no arquivo possuem diversos tamanhos, o arquivo é considerado composto de **registros de tamanho variável**. Um arquivo pode ter registros de tamanho variável por vários motivos:

- Os registros do arquivo são do mesmo tipo de registro, mas um ou mais dos campos são de tamanho variável (**campos de tamanho variável**). Por exemplo, o campo Nome de FUNCIONARIO pode ser um campo de tamanho variável.
- Os registros do arquivo são do mesmo tipo de registro, mas um ou mais dos campos podem ter múltiplos valores para registros individuais; esse campo é chamado de **campo repetitivo**, e um grupo de valores para o campo normalmente é chamado de **grupo repetitivo**.
- Os registros do arquivo são do mesmo tipo de registro, mas um ou mais dos campos são **opcionais**, ou seja, eles podem ter valores para alguns, mas não para todos os registros do arquivo (**campos opcionais**).
- O arquivo contém registros de *tipos de registro diferentes* e, portanto, de tamanho variável (**arquivo misto**). Isso ocorreria se registros relacionados de diferentes tipos fossem *agrupados* (colocados juntos) em blocos de disco; por exemplo, os registros de HISTORICO_ESCOLAR de determinado aluno podem ser colocados após o registro desse ALUNO.

Os registros de FUNCIONARIO de tamanho fixo na Figura 17.5(a) têm um tamanho de registro de 71 bytes. Cada registro tem os mesmos campos, e os tamanhos de campo são fixos, de modo que o sistema pode identificar a posição de byte inicial de cada campo em relação à posição inicial do registro. Isso facilita a localização de valores de campo pelos programas que acessam tais arquivos. Observe que é possível representar um arquivo que logicamente deveria ter registros de tamanho variável como um arquivo de registros de tamanho fixo. Por exemplo, no caso dos campos opcionais, poderíamos ter *cada campo* incluído em *cada registro de arquivo*, mas ar-

Snow666

mazenar um valor especial NULL se não houver valor para esse campo. Para um campo repetitivo, poderíamos alocar tantos espaços em cada registro quanto o *número máximo possível de ocorrências* do campo. De qualquer forma, o espaço é desperdiçado quando certos registros não têm valores para todos os espaços físicos fornecidos em cada registro. Agora, vamos considerar outras opções para formatar registros de um arquivo de registros de tamanho variável.

Para *campos de tamanho variável*, cada registro tem um valor para cada campo, mas não sabemos o tamanho exato de alguns valores de campo. Para determinar os bytes em um registro em particular que representa cada campo, podemos usar caracteres **separadores** especiais (como ? ou % ou $) — que não aparecem em qualquer valor do campo — para terminar os campos de tamanho variável, como mostra a Figura 17.5(b), ou podemos armazenar o tamanho do campo em bytes no próprio registro, antes do valor do campo.

Um arquivo de registros com *campos opcionais* pode ser formatado de várias maneiras. Se o número total de campos para o tipo de registro for grande, mas o número de campos que realmente aparecem em um registro típico for pequeno, podemos incluir em cada registro uma sequência de pares <nome-campo, valor-campo> em vez de apenas os valores de campo. Três tipos de caracteres separadores são usados na Figura 17.5(c), embora pudéssemos usar o mesmo caractere separador para as duas primeiras finalidades — separar o nome do campo do valor do campo e separar um campo do campo seguinte. Uma opção mais prática é atribuir um código curto de **tipo de campo** — digamos, um número inteiro — a cada campo e incluir em cada registro uma sequência de pares <tipo-campo, valor-campo>, em vez de pares <nome-campo, valor-campo>.

Um *campo repetitivo* precisa de um caractere separador para afastar os valores repetitivos do campo e outro caractere separador para indicar o término do campo. Finalmente, para um arquivo que inclui *registros de diferentes tipos*, cada registro é precedido por um indicador de **tipo de registro**. É compreensível que os programas que processam arquivos de registros com tamanho variável — que em geral fazem parte do sistema de arquivos e, portanto, ficam

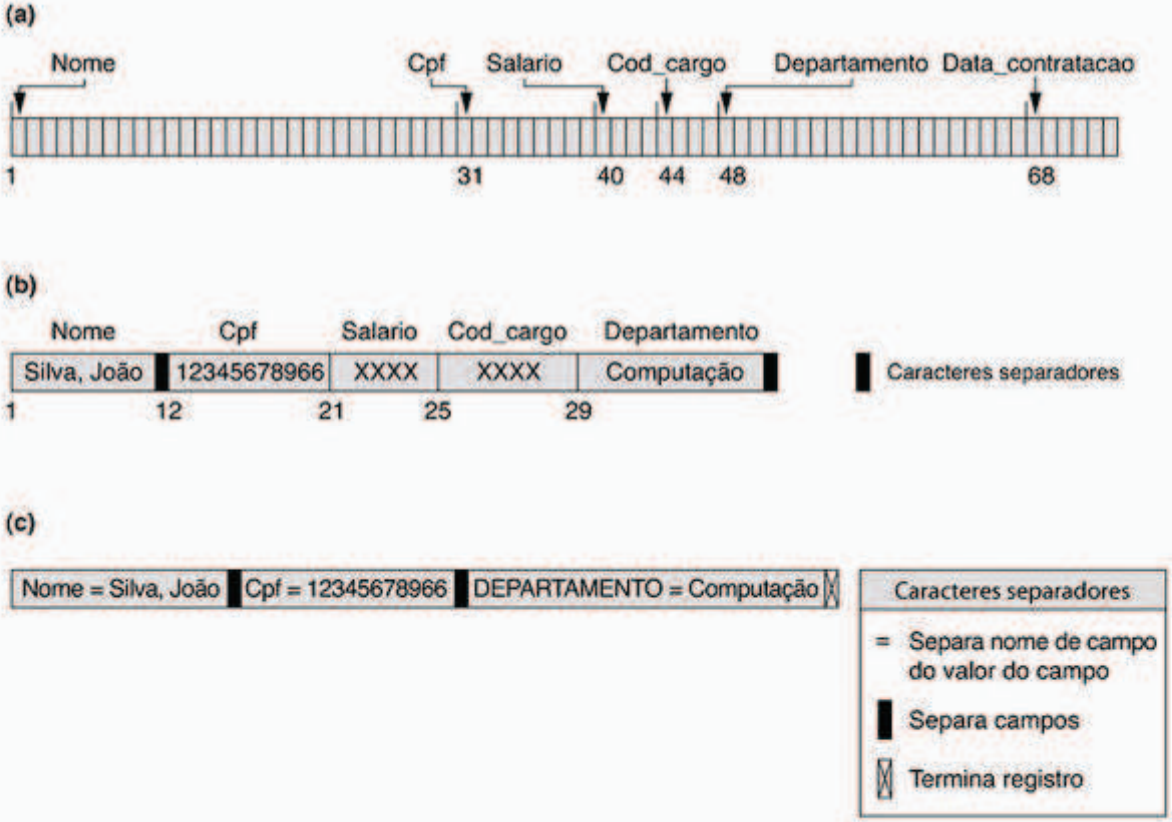

Figura 17.5

Três formatos de armazenamento de registro. (a) Um registro de tamanho fixo com seis campos e tamanho de 71 bytes. (b) Um registro com dois campos de tamanho variável e três campos de tamanho fixo. (c) Um registro de campo variável com três tipos de caracteres separadores.

Snow666

ocultos dos programadores típicos — tenham de ser mais complexos do que aqueles para registros de tamanho fixo, nos quais a posição inicial e o tamanho de cada campo são conhecidos e fixos.[5]

17.4.3 Blocagem de registros e registros espalhados *versus* não espalhados

Os registros de um arquivo precisam ser alocados a blocos de disco, porque um bloco é a *unidade de transferência de dados* entre o disco e a memória. Quando o tamanho do bloco é maior que o tamanho do registro, cada bloco terá diversos registros, embora alguns arquivos possam ter registros excepcionalmente grandes que não cabem em um bloco. Suponha que o tamanho do bloco seja B bytes. Para um arquivo de registros de tamanho fixo, com tamanho de R bytes, sendo $B \geq R$, podemos estabelecer $bfr = \lfloor B/R \rfloor$ registros por bloco, onde o $\lfloor (x) \rfloor$ (*função floor*) *arredonda para baixo* o número x para um inteiro. O valor bfr é chamado de **fator de blocagem** para o arquivo. Em geral, R pode não dividir B exatamente, de modo que temos algum espaço não usado em cada bloco, igual a

$$B - (bfr * R) \text{ bytes}$$

Para aproveitar esse espaço não usado, podemos armazenar parte de um registro em um bloco e o restante em outro. Um **ponteiro** no final do primeiro bloco aponta para o bloco que contém o restante do registro, caso não seja o próximo bloco consecutivo no disco. Essa organização é chamada de **espalhada** porque os registros podem se espalhar por mais de um bloco. Sempre que um registro é maior que um bloco, *temos de* usar uma organização espalhada. Se os registros não puderem atravessar os limites de bloco, a organização é chamada de **não espalhada**. Isso é usado com registros de tamanho fixo tendo $B > R$, pois faz cada registro começar em um local conhecido no bloco, simplificando o processamento do registro. Para registros de tamanho variável, pode-se usar uma organização espalhada ou não espalhada. Se o registro médio for grande, é vantajoso usar o espalhamento para reduzir o espaço perdido em cada bloco. A Figura 17.6 ilustra a organização espalhada *versus* a não espalhada.

Para registros de tamanho variável que usam a organização espalhada, cada bloco pode armazenar um número diferente de registros. Nesse caso, o fator de bloco bfr representa o número *médio* de registros por bloco para o arquivo. Podemos usar bfr para calcular o número de blocos b necessários para um arquivo de r registros:

$$b = \lceil (r/bfr) \rceil \text{ blocos}$$

onde o $\lceil (x)^* \rceil$ (*função ceiling*) arredonda para cima o valor de x até o próximo inteiro.

17.4.4 Alocando blocos de arquivo no disco

Existem várias técnicas-padrão para alocar os blocos de um arquivo no disco. Na **alocação contígua**, os blocos de arquivo são alocados a blocos de disco consecutivos. Isso torna a leitura do arquivo inteiro muito rápida usando o buffering duplo, mas dificulta a expansão do arquivo. Na **alocação ligada**, cada bloco de arquivo contém um ponteiro para o próximo bloco de arquivo. Isso facilita a expansão do arquivo, mas torna sua leitura mais lenta. Uma combinação dos dois aloca **clusters** de blocos de disco consecutivos, e os clusters são ligados. Os clusters às vezes são chamados de **segmentos** ou **extensões de arquivo**. Outra possibilidade é usar a **alocação indexada**, em que um ou mais **blocos de índice** contêm ponteiros para os blocos de arquivo reais. Também é comum usar combinações dessas técnicas.

17.4.5 Cabeçalhos de arquivo

Um **cabeçalho de arquivo** ou **descritor de arquivo** contém informações sobre um arquivo, que são exigi-

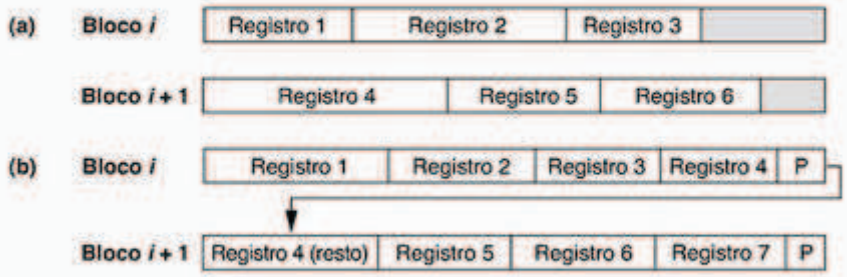

Figura 17.6
Tipos de organização de registro. (a) Não espalhada. (b) Espalhada.

[5] Outros esquemas também são possíveis para representar registros de tamanho variável.

Snow666

das pelos programas do sistema que acessam os registros do arquivo. O cabeçalho inclui informações para determinar os endereços de disco dos blocos de arquivo, bem como para registrar descrições de formato, que podem incluir tamanhos de campo e a ordem dos campos em um registro, para registros não espalhados de tamanho fixo, e códigos de tipo de campo, caracteres separadores e códigos de tipo de registro, para registros de tamanho variável.

Para procurar um registro no disco, um ou mais blocos são copiados para os buffers da memória principal. Os programas então procuram o registro ou registros desejados nos buffers, usando a informação no cabeçalho de arquivo. Se o endereço do bloco que contém o registro desejado não for conhecido, os programas de pesquisa devem realizar uma **pesquisa linear** pelos blocos de arquivo. Cada bloco de arquivo é copiado para um buffer e pesquisado até que o registro seja localizado e todos os blocos do arquivo tenham sido pesquisados com sucesso. Isso pode ser muito demorado para um arquivo grande. O objetivo de uma boa organização de arquivo é localizar o bloco que contém um registro desejado com um número mínimo de transferências de bloco.

17.5 Operações em arquivos

Operações em arquivos costumam ser agrupadas em **operações de recuperação** e **operações de atualização**. O primeiro grupo não muda quaisquer dados no arquivo, apenas localiza certos registros de modo que seus valores de campo possam ser examinados e processados. O segundo muda o arquivo pela inserção ou exclusão de registros, ou pela modificação dos valores de campo. De qualquer forma, podemos ter de **selecionar** um ou mais registros para recuperação, exclusão ou modificação com base em uma **condição de seleção** (ou **condição de filtragem**), que especifica critérios que o registro ou registros desejado(s) deve(m) satisfazer.

Considere um arquivo FUNCIONARIO com os campos Nome, Cpf, Salario, Cod_cargo e Departamento. Uma **condição de seleção simples** pode envolver uma comparação de igualdade em algum valor de campo — por exemplo, (Cpf = '12345678966') ou (Departamento = 'Pesquisa'). Condições mais complexas podem envolver outros tipos de operadores de comparação, como > ou ≥; um exemplo é (Salario ≥ 30.000). O caso geral é ter uma expressão booleana qualquer nos campos do arquivo como condição de seleção.

As operações de pesquisa em arquivos geralmente são baseadas em condições de seleção simples. Uma condição complexa deve ser decomposta pelo SGBD (ou pelo programador) para extrair uma condição simples que pode ser usada para localizar os registros no disco. Cada registro localizado é então verificado para determinar se satisfaz a condição de seleção inteira. Por exemplo, podemos extrair a condição simples (Departamento = 'Pesquisa') da condição complexa ((Salario ≥ 30.000) AND (Departamento = 'Pesquisa'); cada registro satisfazendo (Departamento = 'Pesquisa') é localizado e depois testado para ver se também satisfaz (Salario ≥ 30.000).

Quando vários registros de arquivo satisfazem uma condição de pesquisa, o *primeiro* registro — em relação à sequência física de registros de arquivo — é inicialmente localizado e designado como o **registro atual**. Operações de busca subsequentes começam desse registro e localizam o *próximo* registro no arquivo que satisfaz a condição.

As operações reais para localizar e acessar registros de arquivo variam de um sistema para outro. A seguir, apresentamos um conjunto de operações representativas. Normalmente, programas de alto nível, como programas de software de SGBD, acessam registros usando esses comandos, de modo que às vezes nos referimos a **variáveis de programa** nas descrições a seguir:

- **Open.** Prepara o arquivo para leitura ou gravação. Aloca buffers apropriados (em geral, pelo menos dois) para manter blocos de arquivo do disco, e recupera o cabeçalho do arquivo. Define o ponteiro de arquivo para o início do arquivo.
- **Reset.** Define o ponteiro do arquivo aberto para o início do arquivo.
- **Find (ou Locate).** Procura o primeiro registro que satisfaça uma condição de pesquisa. Transfere o bloco que contém esse registro para o buffer da memória principal (se ainda não estiver lá). O ponteiro de arquivo aponta para o registro no buffer e este se torna o *registro atual*. Às vezes, diferentes verbos são usados para indicar se o registro localizado deve ser lido ou atualizado.
- **Read (ou Get).** Copia o registro atual do buffer para uma variável de programa no programa do usuário. Esse comando também pode avançar o ponteiro do registro atual para o próximo registro no arquivo, que pode precisar ler o próximo bloco de arquivo do disco.
- **FindNext.** Procura o próximo registro no arquivo que satisfaz a condição de pesquisa.

Snow666

Transfere o bloco que contém esse registro para um buffer da memória principal (se ainda não estiver lá). O registro está localizado no buffer e torna-se o registro atual. Várias formas de FindNext (por exemplo, encontrar próximo registro dentro de um registro pai atual, encontrar próximo registro de determinado tipo ou encontrar próximo registro em que uma condição complexa é atendida) estão disponíveis em SGBDs legados com base nos modelos hierárquico e de rede.

- **Delete.** Exclui o registro atual e (no fim) atualiza o arquivo no disco para refletir a exclusão.
- **Modify.** Modifica alguns valores de campo para o registro atual e (no fim) atualiza o arquivo no disco para refletir a modificação.
- **Insert.** Insere um novo registro no arquivo ao localizar o bloco onde o registro deve ser inserido, transferindo esse bloco para um buffer da memória principal (se ainda não estiver lá), gravando o registro no buffer e (no fim) gravando o buffer em disco para refletir a inserção.
- **Close.** Completa o acesso ao arquivo liberando os buffers e realizando quaisquer outras operações de limpeza necessárias.

Estas (exceto por Open e Close) são chamadas operações de **um registro por vez**, pois cada uma se aplica a um único registro. É possível resumir as operações Find, FindNext e Read em uma única operação, Scan, cuja descrição é a seguinte:

- **Scan.** Se o arquivo já tiver sido aberto ou reiniciado, *Scan* retorna o primeiro registro; caso contrário, ele retorna o próximo registro. Se uma condição for especificada com a operação, o registro retornado é o primeiro ou o próximo registro que satisfaz a condição.

Em sistemas de banco de dados, operações adicionais de nível mais alto, de **um conjunto de cada vez**, podem ser aplicadas a um arquivo. Alguns exemplos são os seguintes:

- **FindAll.** Localiza *todos* os registros no arquivo que satisfazem uma condição de pesquisa.
- **Find** (ou **Locate**) ***n***. Procura o primeiro registro que satisfaz uma condição de pesquisa e depois continua a localizar os próximos $n - 1$ registros que satisfazem a mesma condição. Transfere os blocos que contém os n registros para o buffer da memória principal (se ainda não estiverem lá).
- **FindOrdered.** Recupera todos os registros no arquivo em alguma ordem especificada.
- **Reorganize.** Inicia o processo de reorganização. Conforme veremos, algumas organizações de arquivo exigem reorganização periódica. Um exemplo é reordenar os registros do arquivo classificando-os em um campo especificado.

Neste ponto, vale a pena notar a diferença entre os termos *organização de arquivo* e *método de acesso*. Uma **organização de arquivo** refere-se à organização dos dados de um arquivo em registros, blocos e estruturas de acesso; isso inclui o modo como registros e blocos são colocados no meio de armazenamento e interligados. Um **método de acesso**, por sua vez, oferece um grupo de operações — como aquelas listadas anteriormente — que podem ser aplicadas a um arquivo. Em geral, é possível aplicar vários métodos de acesso a uma organização de arquivo. Alguns métodos de acesso, porém, só podem ser aplicados a arquivos organizados de certas maneiras. Por exemplo, não podemos aplicar um método de acesso indexado a um arquivo sem um índice (ver Capítulo 18).

Normalmente, esperamos usar algumas condições de pesquisa mais do que outras. Certos arquivos podem ser **estáticos**, significando que operações de atualização raramente são realizadas; outros arquivos, mais **dinâmicos**, podem mudar com frequência, de modo que as operações de atualizações são constantemente aplicadas a eles. Uma organização de arquivo bem-sucedida deve realizar, do modo mais eficiente possível, as operações que esperamos *aplicar frequentemente* ao arquivo. Por exemplo, considere o arquivo FUNCIONARIO, mostrado na Figura 17.5(a), que armazena os registros para os funcionários ativos em uma empresa. Esperamos inserir registros (quando os funcionários são contratados), excluir registros (quando os funcionários saem da empresa) e modificar registros (por exemplo, quando o salário ou cargo de um funcionário é alterado). A exclusão ou modificação de um registro requer uma condição de seleção para identificar um registro em particular ou conjunto de registros. A leitura de um ou mais registros também requer uma condição de seleção.

Se os usuários esperam principalmente aplicar uma condição de pesquisa com base no Cpf, o projetista precisa escolher uma organização de arquivo que facilita a localização de um registro, dado seu valor de Cpf. Isso pode envolver a ordenação física dos registros pelo valor do Cpf ou a definição de um índice em Cpf (ver Capítulo 18). Suponha que uma segunda aplicação utilize o arquivo para gerar contracheques de funcionários e exija que os contracheques sejam agrupados por departamento. Para essa aplicação, é melhor ordenar os registros de funcionários por departamento e depois por

Snow666

nome dentro de cada departamento. O agrupamento de registros em blocos e a organização de blocos em cilindros agora seriam diferentes. Porém, esse arranjo entra em conflito com a ordenação dos registros por valores de Cpf. Se as duas aplicações são importantes, o projetista deve escolher uma organização que permita que as duas operações sejam realizadas de modo eficiente. Infelizmente, em muitos casos, uma única organização não permite que todas as operações necessárias em um arquivo sejam implementadas de maneira eficiente. Isso requer que seja escolhido um meio-termo que leve em conta a importância esperada e a mistura de operações de leitura e recuperação.

Nas próximas seções e no Capítulo 18, discutimos métodos para organizar registros de um arquivo no disco. Várias técnicas gerais, como ordenação, hashing e indexação, são usadas para criar métodos de acesso. Além disso, diversas técnicas gerais para lidar com inserções e exclusões trabalham com muitas organizações de arquivo.

17.6 Arquivos de registros desordenados (arquivos de heap)

Neste tipo de organização mais simples e mais básico, os registros são arquivados na ordem em que são inseridos, de modo que novos registros são inseridos ao final do arquivo. Essa organização é chamada **arquivo de heap** ou **pilha**.[6] Normalmente, ela é usada com caminhos de acesso adicionais, como os índices secundários discutidos no Capítulo 18. Ela também é usada para coletar e armazenar registros de dados para uso futuro.

A inserção de um novo registro é muito *eficiente*. O último bloco de disco do arquivo é copiado para um buffer, o novo registro é acrescentado e o bloco é então **regravado** de volta no disco. O endereço do último bloco de arquivo é mantido no cabeçalho do arquivo. No entanto, procurar um registro usando qualquer condição de pesquisa envolve uma **pesquisa linear** pelo bloco de arquivo por bloco — um procedimento dispendioso. Se apenas um registro satisfizer a condição de pesquisa, então, na média, um programa lerá a memória e pesquisará metade dos blocos de arquivo antes de encontrar o registro. Para um arquivo de b blocos, isso exige pesquisar $(b/2)$ blocos, na média. Se nenhum registro ou vários registros satisfizerem a condição de pesquisa, o programa deve ler e pesquisar todos os b blocos no arquivo.

Para excluir um registro, um programa deve primeiro encontrar seu bloco, copiá-lo para um buffer, excluir o registro do buffer e, finalmente, **regravar o bloco** de volta ao disco. Isso deixa um espaço livre no bloco de disco. A exclusão de um grande número de registros dessa maneira resulta em espaço de armazenamento desperdiçado. Outra técnica usada para exclusão de registro é ter um byte ou bit extra, chamado **marcador de exclusão**, armazenado em cada registro. Um registro é excluído ao se definir o marcador de exclusão com determinado valor. Um valor diferente para o marcador indica um registro válido (não excluído). Os programas de pesquisa consideram apenas registros válidos em um bloco quando realizam sua busca. Essas duas técnicas de exclusão exigem **reorganização** periódica do arquivo para retomar o espaço não usado dos registros excluídos. Durante a reorganização, os blocos de arquivo são acessados de maneira consecutiva, e os registros são compactados pela remoção de registros excluídos. Depois dessa reorganização, os blocos são preenchidos até a capacidade mais uma vez. Outra possibilidade é usar o espaço dos registros excluídos ao inserir novos registros, embora isso exija uma manutenção extra para se manter informado sobre os locais vazios.

Podemos usar a organização espalhada ou não espalhada para um arquivo desordenado, e ela pode ser utilizada com registros de tamanho fixo ou variável. A modificação de um registro de tamanho variável pode exigir a exclusão do registro antigo e a inserção de um registro modificado, pois o registro modificado pode não se encaixar em seu antigo espaço no disco.

Para ler todos os registros na ordem dos valores de algum campo, criamos uma cópia classificada do arquivo. A classificação é uma operação cara para um arquivo de disco grande, e técnicas especiais para **classificação externa** são utilizadas (ver Capítulo 19).

Para um arquivo de *registros de tamanho fixo* desordenados usando *blocos não espalhados* e *alocação contígua*, é simples acessar qualquer registro por sua **posição** no arquivo. Se os registros de arquivo forem numerados com 0, 1, 2, ..., $r - 1$ e os registros em cada bloco forem numerados com 0, 1, ..., $bfr - 1$, onde bfr é o fator de bloco, então o i-ésimo registro do arquivo está localizado no bloco $\lfloor (i/bfr) \rfloor$ e é o $(i \bmod bfr)^{\text{o}}$ registro nesse bloco. Tal arquivo costuma ser chamado de **arquivo relativo** ou **direto**, pois os registros podem ser facilmente acessados por suas posições relativas. O acesso a um registro por sua posição não ajuda a localizar um registro com base em uma condição de busca; contudo, ele facilita a construção de caminhos de acesso no arquivo, como os índices discutidos no Capítulo 18.

[6] Às vezes, essa organização é chamada de **arquivo sequencial**.

Snow666

17.7 Arquivos de registros ordenados (arquivos classificados)

Podemos ordenar fisicamente os registros de um arquivo no disco com base nos valores de um de seus campos — chamado de **campo de ordenação**. Isso leva a um arquivo **ordenado** ou **sequencial**.[7] Se o campo de ordenação também for um **campo-chave** do arquivo — um campo com garantias de ter um valor exclusivo em cada registro —, então o campo é chamado de **chave de ordenação** para o arquivo. A Figura 17.7 mostra um arquivo ordenado com Nome como campo-chave de ordenação (supondo que os funcionários têm nomes distintos).

Os registros ordenados têm algumas vantagens em relação aos arquivos desordenados. Primeiro, a leitura dos registros na ordem dos valores da chave de ordenação torna-se extremamente eficiente porque nenhuma classificação é necessária. Segundo, encontrar o próximo registro com base no atual na ordem da chave de ordenação em geral não requer acessos de bloco adicionais porque o próximo registro está no mesmo bloco do atual (a menos que o registro atual seja o último no bloco). Terceiro, o uso de uma condição de pesquisa baseada no valor de um campo-chave de ordenação resulta em acesso mais rápido quando a técnica de pesquisa binária é usada, o que constitui uma melhoria em relação às pesquisas lineares, embora normalmente isso não seja utilizado para arquivos de disco. Os arquivos ordenados estão em blocos e armazenados em cilindros contíguos para minimizar o tempo de busca.

Uma **pesquisa binária** por arquivos de disco pode ser feita nos blocos, em vez de nos registros. Suponha que o arquivo tenha b blocos numerados com 1, 2, ..., b; os registros são ordenados por valor crescente de seu campo-chave de ordenação; e estamos procurando um registro cujo valor do campo-chave de ordenação seja K. Supondo que os endereços de disco dos blocos de arquivo estejam disponíveis no cabeçalho de arquivo, a pesquisa binária pode ser descrita pelo Algoritmo 17.1. Uma pesquisa binária normalmente acessa $\log_2(b)$ blocos, não importando se o registro foi localizado ou não — uma melhoria em relação às pesquisas lineares, onde, na média, $(b/2)$ blocos são acessados quando o registro é encontrado e b blocos são acessados quando o registro não é encontrado.

Algoritmo 17.1. Pesquisa binária em uma chave de ordenação de um arquivo de disco

```
l ← 1; u ← b; (* b é o número de blocos de arquivo *)
enquanto (u ≥ l ) faca
    inicio i ← (l + u) div 2;
    leia bloco i do arquivo para o buffer;
    se K < (valor do campo-chave de ordenação
    do primeiro registro no bloco i)
        então u ← i – 1
    senão se K > (valor do campo-chave de or-
    denação do último registro no bloco i )
        então l ← i + 1
    senão se o registro com valor do campo-cha-
    ve de ordenação = K está no buffer
        então vai para found
    senão vai para notfound;
    fim;
vai para notfound;
```

Um critério de pesquisa envolvendo as condições >, <, ≥ e ≤ no campo de ordenação é muito eficiente, pois a ordenação física dos registros significa que todos os registros que satisfazem a condição são contíguos no arquivo. Por exemplo, com relação à Figura 17.7, se o critério de pesquisa for (Nome < 'G') — onde < significa *alfabeticamente antes de* —, os registros que satisfazem o critério de pesquisa são aqueles do início do arquivo até o primeiro registro que tem um valor Nome começando com a letra 'G'.

A ordenação não oferece quaisquer vantagens para o acesso aleatório ou ordenado dos registros com base nos valores dos outros *campos não ordenados* do arquivo. Nesses casos, realizamos uma pesquisa linear para acesso aleatório. Para acessar os registros em ordem baseados no campo não ordenados, é necessário criar outra cópia ordenada — em uma ordem diferente — do arquivo.

A inserção e a exclusão de registros são operações dispendiosas para um arquivo ordenado, pois os registros devem permanecer fisicamente ordenados. Para inserir um registro, temos de encontrar sua posição correta no arquivo, com base no valor de seu campo de ordenação, e depois criar espaço no arquivo para inserir o registro nessa posição. Para um arquivo grande, isso pode ser muito demorado porque, na média, metade dos registros no arquivo precisa ser movida para criar espaço para o novo registro. Isso significa que metade dos blocos de arquivo deve ser lida e regravada depois que os registros forem movidos entre eles. Para a exclusão de registro, o problema é menos grave se os marcadores de exclusão e a reorganização periódica forem usados.

[7] O termo *arquivo sequencial* também tem sido usado para se referir aos arquivos desordenados, embora seja mais apropriado para arquivos ordenados.

Snow666

	Nome	Cpf	Data_nascimento	Cargo	Salario	Sexo
Bloco 1	Aaron, Eduardo					
	Abílio, Diana					
			⋮			
	Acosta, Marcos					
Bloco 2	Adams, João					
	Adams, Roberto					
			⋮			
	Akers, Janete					
Bloco 3	Alexandre, Eduardo					
	Alfredo, Roberto					
			⋮			
	Allen, Samuel					
Bloco 4	Allen, Tiago					
	Anderson, Kely					
			⋮			
	Anderson, Joel					
Bloco 5	Anderson, Isaac					
	Angeli, José					
			⋮			
	Anita, Sueli					
Bloco 6	Arnoldo, Marcelo					
	Arnoldo, Estevan					
			⋮			
	Atílio, Timóteo					
			⋮			
Bloco *n*–1	Wanderley, Jaime					
	Wesley, Ronaldo					
			⋮			
	Wong, Manuel					
Bloco *n*	Wong, Pâmela					
	Wuang, Charles					
			⋮			
	Zimmer, André					

Figura 17.7
Alguns blocos de um arquivo ordenado (sequencial) de registros de FUNCIONARIO com Nome como campo-chave de ordenação.

Uma opção para tornar a inserção mais eficiente é manter algum espaço não usado em cada bloco para novos registros. Porém, quando esse espaço é totalmente utilizado, o problema original reaparece. Outro método frequentemente empregado é criar um arquivo *desordenado* temporário, chamado arquivo de **overflow** ou **transação**. Com essa técnica, o arquivo ordenado real é chamado de arquivo **principal** ou **mestre**. Novos registros são inseridos no final do arquivo de overflow, em vez de em sua posição correta no arquivo principal. De tempos em tempos, o arquivo de overflow é classificado e mesclado ao arquivo mestre durante a reorganização do arquivo. A inserção torna-se muito eficiente, mas ao custo de maior complexidade no algoritmo de pesquisa. O arquivo de overflow precisa ser pesquisado usando uma pesquisa linear se, após a pesquisa binária, o registro não for encontrado no arquivo principal. Para

Snow666

aplicações que não exigem a informação mais atualizada, os registros de overflow podem ser ignorados durante uma pesquisa.

A modificação do valor de um campo de um registro depende de dois fatores: a condição de pesquisa para localizar o registro e o campo a ser modificado. Se a condição de pesquisa envolver o campo da chave de ordenação, podemos localizar o registro com uma pesquisa binária; caso contrário, temos de realizar uma pesquisa linear. Um campo não ordenado pode ser modificado ao alterar o registro e regravá-lo no mesmo local físico no disco — considerando registros de tamanho fixo. A modificação do campo de ordenação significa que o registro pode alterar sua posição no arquivo. Isso requer a exclusão do registro antigo, seguida pela inserção do registro modificado.

A leitura dos registros do arquivo na ordem do campo de ordenação é muito eficiente se ignorarmos os registros no overflow, pois os blocos podem ser lidos consecutivamente ao usar o buffering duplo. Para incluir os registros no overflow, temos de mesclá-los em suas posições atuais; nesse caso, primeiro podemos reorganizar o arquivo, e depois ler seus blocos na sequência. Para reorganizar o arquivo, classificamos os registros no arquivo de overflow e, depois, os mesclamos com o arquivo mestre. Os registros marcados para exclusão são removidos durante a reorganização.

A Tabela 17.2 resume o tempo de acesso médio em acessos de bloco para encontrar um registro específico em um arquivo com b blocos.

Os arquivos ordenados raramente são usados em aplicações de banco de dados, a menos que um caminho de acesso adicional, chamado **índice primário**, seja utilizado; isso resulta em um **arquivo sequencial-indexado**. Isso melhora ainda mais o tempo de acesso aleatório ao campo-chave de ordenação. (Discutiremos índices no Capítulo 18.) Se o atributo de ordenação não for uma chave, o arquivo é chamado de **arquivo agrupado**.

17.8 Técnicas de hashing

Outro tipo de organização de arquivo principal é baseado no hashing, que oferece acesso muito rápido aos registros sob certas condições de pesquisa. Essa organização costuma ser chamada de **arquivo de hash**.[8] A condição de pesquisa precisa ser uma condição de igualdade em um único campo, chamado **campo de hash**. Na maior parte dos casos, o campo de hash também é um campo-chave do arquivo, quando ele é chamado de **chave hash**. A ideia por trás do hashing é oferecer uma função h, chamada **função de hash** ou **função de randomização**, que é aplicada ao valor do campo de hash de um registro e gera o *endereço* do bloco de disco em que o registro está armazenado. Uma pesquisa pelo registro no bloco pode ser executada em um buffer da memória principal. Para a maioria dos registros, só precisamos de um acesso de único bloco para recuperar esse registro.

O hashing também é utilizado como uma estrutura de pesquisa interna em um programa sempre que um grupo de registros é acessado exclusivamente pelo uso do valor de um campo. Descrevemos o uso do hashing para arquivos internos na Seção 17.8.1; depois, mostramos como ele é modificado para armazenar arquivos externos no disco na Seção 17.8.2. Na Seção 17.8.3, discutimos técnicas para estender o hashing a arquivos dinamicamente crescentes.

17.8.1 Hashing interno

Para arquivos internos, o hashing normalmente é implementado como uma **tabela de hash** por meio do uso de um array de registros. Suponha que o intervalo de índice de array seja de 0 a $M - 1$, como mostra a Figura 17.8(a). Então, temos M **slots** cujos endereços correspondem aos índices de array. Escolhemos uma função de hash que transforma o valor de campo de hash em um inteiro entre 0 e $M - 1$. Uma função de hash comum é $h(K) = K \bmod M$, que retorna o resto de um valor de campo de hash inteiro

Tabela 17.2
Tempos de acesso médios para um arquivo de b blocos sob organizações de arquivo básicas.

Tipo de organização	Método de acesso/pesquisa	Média de blocos para acessar um registro específico
Heap (não ordenado)	Varredura sequencial (pesquisa linear)	$b/2$
Ordenado	Varredura sequencial	$b/2$
Ordenado	Pesquisa binária	$\log_2 b$

[8] Um arquivo de hash também é chamado de *arquivo direto*.

Snow666

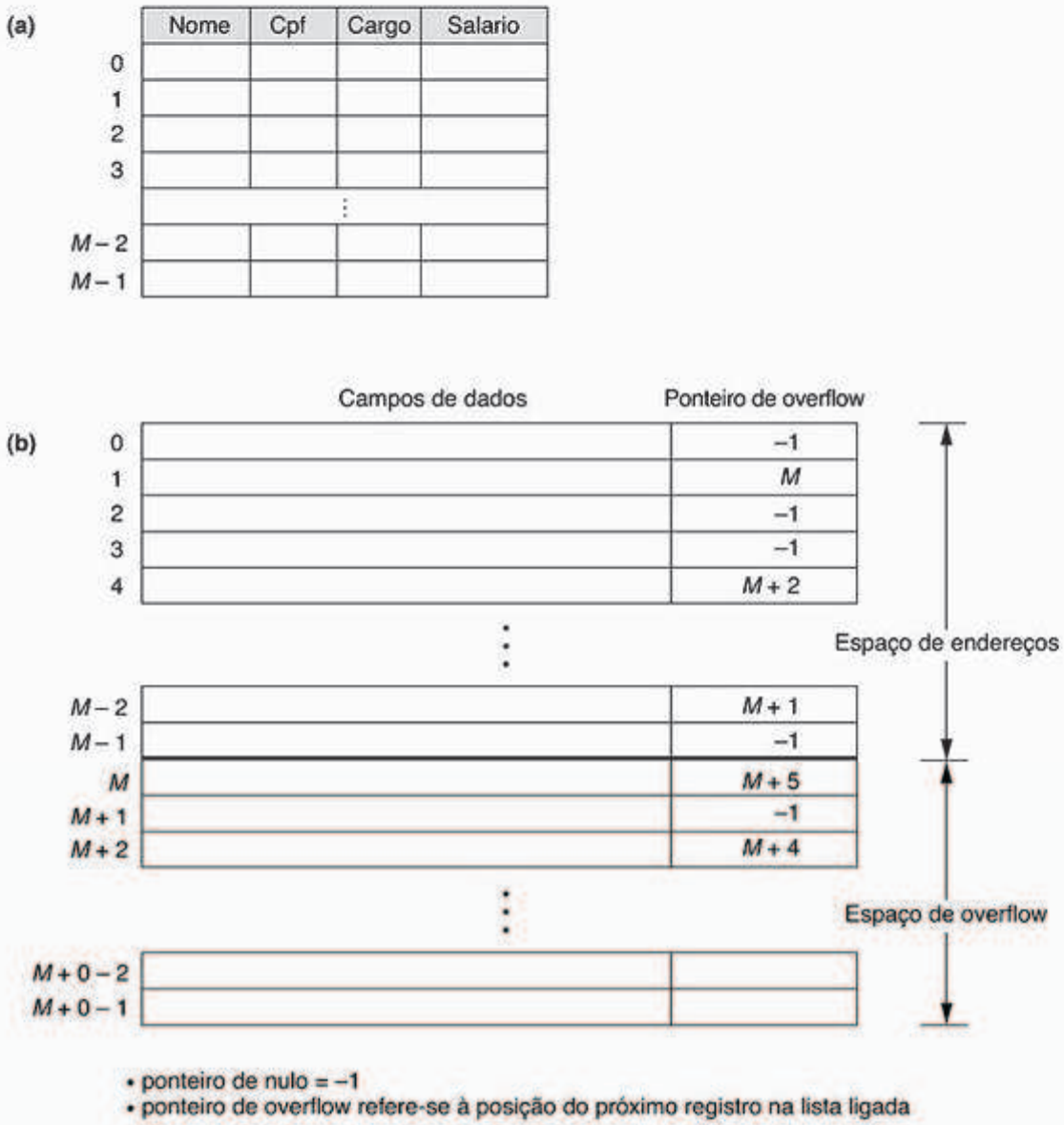

Figura 17.8

Estruturas de dados de hashing interno. (a) Array de *M* posições para uso no hashing interno. (b) Resolução de colisão ao encadear registros.

K após a divisão por *M*; esse valor é então usado para o endereço do registro.

Os valores de campo de hash não inteiros podem ser transformados em inteiros antes que a função mod seja aplicada. Para cadeias de caracteres, os códigos numéricos (ASCII) associados aos caracteres podem ser usados na transformação — por exemplo, ao multiplicar esses valores de código. Para um campo de hash cujo tipo de dado é uma string de 20 caracteres, o Algoritmo 17.2(a) pode ser utilizado para calcular o endereço de hash. Assumimos que a função de código retorna o código numérico de um caractere e que recebemos um valor de campo de hash *K* do tipo *K*: *array* [1..20] *of char* (em Pascal) ou *char K*[20] (em C).

Algoritmo 17.2. Dois algoritmos de hashing simples: (a) aplicando a função de hash mod a uma cadeia de caracteres K. (b) Resolução de colisão por endereçamento aberto.

(a) *temp* ← 1;

para *i* ← 1 até 20 faça *temp* ← *temp* * code(*K*[*i*]) mod *M* ;

endereco_hash ← *temp* mod *M*;

(b) *i* ← *endereco_hash*(*K*); *a* ← *i*;

se local *i* está ocupado

entao inicio *i* ← (*i* + 1) mod *M*;

enquanto (*i* ≠ *a*) e local *i* está ocupado

Snow666

faça $i \leftarrow (i + 1) \bmod M$;
se $(i = a)$ então todas as posições estão cheias
se não *novo_endereco_hash* $\leftarrow i$;
fim;

Outras funções de hashing podem ser usadas. Uma técnica, chamada **desdobramento**, envolve aplicar uma função aritmética, como a *adição*, ou uma função lógica, como o *or exclusivo* a diferentes partes do valor do campo de hash para calcular o endereço de hash (por exemplo, com um espaço de endereços de 0 a 999 para armazenar 1.000 chaves, uma chave de seis dígitos 235469 pode ser desdobrada e armazenada no endereço: (235+964) mod 1000 = 199). Outra técnica envolve escolher alguns dígitos do valor do campo de hash — por exemplo, terceiro, quinto e oitavo dígitos — para formar o endereço de hash (por exemplo, armazenando 1.000 funcionários com números de CPF de 11 dígitos em um arquivo de hash com 1.000 posições daria ao número de CPF 301--678-923-51 um valor de hash de 172 por essa função de hash).[9] O problema com a maioria das funções de hashing é que elas não garantem que valores distintos terão endereços de hash distintos, pois o **espaço do campo de hash** — o número de valores possíveis que um campo de hash pode ter — normalmente é muito maior do que o **espaço de endereços** — o número de endereços disponíveis para registros. A função de hashing mapeia o espaço do campo de hash ao espaço de endereços.

Uma **colisão** ocorre quando o valor do campo de hash de um registro que está sendo inserido é calculado como um endereço que já contém um registro diferente. Nessa situação, temos de inserir o novo registro em alguma outra posição, pois o endereço de hash está ocupado. O processo de localizar outra posição é chamado de **resolução de colisão**. Existem vários métodos para resolução de colisão, incluindo os seguintes:

- **Endereçamento aberto.** Partindo da posição ocupada especificada pelo endereço de hash, o programa verifica as posições subsequentes em ordem, até que uma posição não usada (vazia) seja encontrada. O Algoritmo 17.2(b) pode ser usado para essa finalidade.
- **Encadeamento.** Para esse método, vários locais de overflow são mantidos, normalmente estendendo o array com uma série de posições de overflow. Além disso, um campo de ponteiro é acrescentado ao local de cada registro. Uma colisão é resolvida ao colocar o novo registro em um local de overflow não usado e definir o ponteiro do local do endereço de hash ocupado como o endereço desse local de overflow. Portanto, é mantida uma lista ligada de registros de overflow para cada endereço de hash, como mostra a Figura 17.8(b).
- **Hashing múltiplo.** O programa aplica uma segunda função de hash se a primeira resultar em uma colisão. Se houver outra colisão, o programa utiliza o endereçamento aberto ou aplica uma terceira função de hash e, depois, usa o endereçamento aberto, se necessário.

Cada método de resolução de colisão requer os próprios algoritmos para inserção, recuperação e exclusão de registros. Os algoritmos para encadeamento são os mais simples. Os algoritmos de exclusão para endereçamento aberto são mais complicados. Os livros-texto sobre estruturas de dados abordam algoritmos de hashing interno com mais detalhes.

O objetivo de uma boa função de hashing é distribuir os registros de maneira uniforme pelo espaço de endereços de modo a minimizar as colisões enquanto não deixam muitos locais não usados. A simulação e os estudos de análise mostraram que normalmente é melhor manter uma tabela de hash entre 70 e 90 por cento cheia, de modo que o número de colisões permaneça baixo e não desperdicemos muito espaço. Logo, se esperamos ter r registros para armazenar na tabela, devemos escolher M locais para o espaço de endereços, tal que (r/M) esteja entre 0,7 e 0,9. Também pode ser útil escolher um número primo para M, pois já foi demonstrado que isso distribui melhor os endereços de hash pelo espaço de endereços quando a função de hashing mod é utilizada. Outras funções de hash podem exigir que M seja uma potência de 2.

17.8.2 Hashing externo para arquivos de disco

O hashing para arquivos de disco é chamado de **hashing externo**. Para se ajustar às características do armazenamento de disco, o espaço de endereços de destino é feito em **buckets**, cada um mantendo vários registros. Um bucket é um bloco de disco ou um cluster de blocos de disco contíguos. A função de hashing mapeia uma chave em um número de bucket relativo, em vez de atribuir um endereço de bloco absoluto ao bucket. Uma tabela mantida no cabeçalho do arquivo converte o número do bucket para o endereço do bloco de disco correspondente, conforme ilustra a Figura 17.9.

O problema de colisão é menos sério com os buckets porque, independentemente de quantos regis-

[9] Uma discussão detalhada das funções de hashing está fora do escopo de nossa apresentação.

Snow666

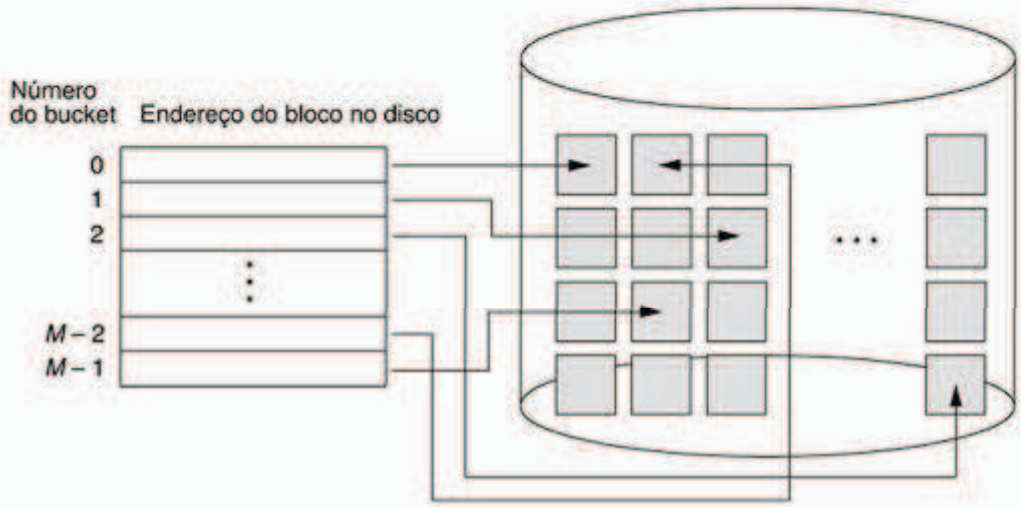

Figura 17.9
Correspondendo números de bucket a endereços de bloco de disco.

tros possam caber em um bucket, eles podem ser definidos por hashing ao mesmo bucket sem causar problemas. Porém, temos de prever o caso em que um bucket está cheio até sua capacidade e um novo registro a ser inserido tem um hash para esse bucket. Podemos usar uma variação do encadeamento em que um ponteiro é mantido em cada bucket para uma lista ligada de registros de overflow para o bucket, como mostra a Figura 17.10. Os ponteiros na lista ligada devem ser **ponteiros de registro**, que incluem um endereço de bloco e uma posição de registro relativa no bloco.

O hashing oferece o acesso mais rápido possível para recuperar um registro qualquer dado o valor de seu campo de hash. Embora a maioria das boas funções de hash não mantenha registros na ordem dos valores de campo de hash, algumas funções — chamadas **preservação de ordem** — o fazem. Um exemplo simples de uma função de hash de preservação de ordem é usar os três dígitos mais à esquerda de um número de fatura para gerar um endereço de bucket como um endereço de hash e manter os registros classificados por número de fatura em cada bucket. Outro exemplo é usar uma chave hash inteira diretamente como um índice para um arquivo relativo, se os valores de chave hash preencherem determinado intervalo; por exemplo, se os números de funcionário em uma empresa forem definidos como 1, 2, 3, ... até o número total de funcionários, podemos usar a função de hash de identidade que mantém a ordem. Infelizmente, isso só funciona se as chaves forem geradas em ordem por alguma aplicação.

O esquema de hashing descrito até aqui é chamado de **hashing estático**, pois um número fixo de buckets M é alocado. Essa pode ser uma desvantagem séria para arquivos dinâmicos. Suponha que aloquemos M buckets para o espaço de endereços e deixemos m como o número máximo de registros que podem caber em um bucket; então, no máximo $(m * M)$ registros caberão no espaço alocado. Se o número de registros for substancialmente menor que $(m * M)$, ficamos com muito espaço não usado. Por sua vez, se o número de registros aumentar para muito mais do que $(m * M)$, várias colisões acontecerão, e a recuperação será mais lenta devido às longas listas de registros de overflow. Em ambos os casos, podemos ter de alterar o número de blocos M alocados e depois usar uma nova função de hashing (com base no novo valor de M) para redistribuir os registros. Essas reorganizações podem ser muito demoradas para arquivos grandes. Organizações de arquivo dinâmico mais recentes, baseadas em hashing, permitem que o número de buckets varie dinamicamente apenas com a reorganização localizada (ver Seção 17.8.3).

Ao usar o hashing externo, a busca por um registro, dado um valor de algum campo diferente do campo de hash, é tão dispendiosa quanto no caso de um arquivo desordenado. A exclusão de registro pode ser implementada pela remoção do registro de seu bucket. Se o bucket tiver uma cadeia de overflow, podemos mover um dos registros de overflow para o bucket e substituir o registro excluído. Se o registro a ser excluído já estiver em overflow, simplesmente o removemos da lista ligada. Observe que a remoção de um registro de overflow implica que devemos registrar a posições vazias no overflow. Isso é feito facilmente pela manutenção de uma lista interligada de locais de overflow não usados.

A modificação do valor de campo de um registro especificado depende de dois fatores: da condição de pesquisa para localizar esse registro específico e do campo a ser modificado. Se a condição de pesquisa for uma comparação de igualdade no campo de hash, podemos localizar o registro de modo eficiente

Snow666

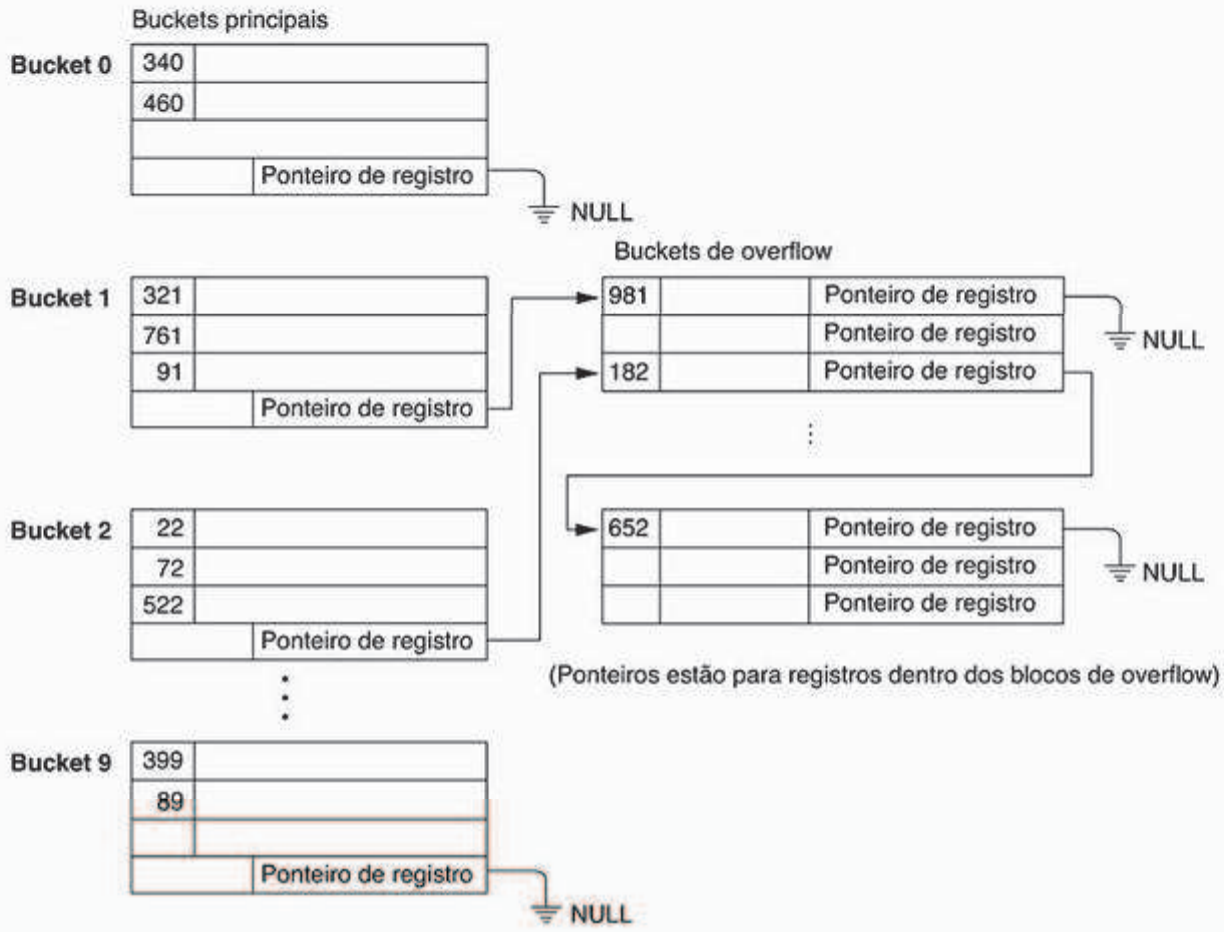

Figura 17.10
Tratamento de overflow para buckets por encadeamento.

usando a função de hashing; caso contrário, temos de realizar uma pesquisa linear. Um campo não de hash pode ser modificado pela mudança do registro e sua regravação no mesmo bucket. A modificação do campo de hash significa que o registro pode se mover para outro bucket, o que exige a exclusão do registro antigo seguida pela inserção do registro modificado.

17.8.3 Técnicas de hashing que permitem a expansão dinâmica do arquivo

Uma grande desvantagem do esquema de hashing *estático* que acabamos de ver é que o espaço de endereços de hash é fixo. Logo, é difícil expandir ou encolher o arquivo dinamicamente. Os esquemas descritos nesta seção tentam solucionar essa situação. O primeiro esquema — o hashing extensível — armazena uma estrutura de acesso além do arquivo e, portanto, é semelhante à indexação (ver Capítulo 18). A principal diferença é que a estrutura de acesso se baseia nos valores que resultam após a aplicação da função de hash ao campo de pesquisa. Na indexação, a estrutura de acesso é baseada nos valores do próprio campo de pesquisa. A segunda técnica, chamada hashing linear, não requer estruturas de acesso adicionais. Outro esquema, chamado **hashing dinâmico**, usa uma estrutura de acesso baseada em estruturas de dados de árvore binária.

Esses esquemas de hashing tiram proveito do fato de que o resultado da aplicação de uma função de hashing é um inteiro não negativo e, portanto, pode ser representado como um número binário. A estrutura de acesso é feita com base na **representação binária** do resultado da função de hashing, que é uma string de **bits**. Chamamos isso de **valor de hash** de um registro. Os registros são distribuídos entre buckets com base nos valores dos *bits iniciais* em seus valores de hash.

Hashing extensível. No hashing extensível, um tipo de diretório — um array de 2^d endereços de bucket — é mantido, onde d é chamado de **profundidade global** do diretório. O valor inteiro correspondente aos primeiros d' bits (ordem alta) de um valor de hash é utilizado como um índice para o array para determinar uma entrada de diretório, e o endereço nessa entrada

Snow666

determina o bucket em que os registros correspondentes são armazenados. Porém, não é preciso haver um bucket distinto para cada um dos 2^d locais de diretório. Vários locais de diretório com os mesmos primeiros d' bits para seus valores de hash podem conter o mesmo endereço de bucket se todos os registros que possuem hash para esses locais couberem em um único bucket. Uma **profundidade local** d' — armazenada com cada bucket — especifica o número de bits em que os conteúdos dos buckets são baseados. A Figura 17.11 mostra um diretório com profundidade global $d = 3$.

O valor de d pode ser aumentado ou diminuído em um de cada vez, dobrando ou reduzindo à metade, o número de entradas no array do diretório. Dobrar é necessário se um bucket, cuja profundidade local d' é igual à profundidade global d, estourar. Reduzir à metade ocorre se $d > d'$ para todos os buckets após ocorrer algumas exclusões. A maioria das recuperações de registro exige dois acessos de bloco — um para o diretório e outro para o bucket.

Para ilustrar a divisão de bucket, suponha que um novo registro inserido cause overflow no bucket cujos valores de hash começam com 01 — o terceiro bucket na Figura 17.11. Os registros serão distribuídos entre dois buckets: o primeiro contém todos os registros cujos valores de hash começam com 010, e o segundo, todos aqueles cujos valores de hash começam com 011. Agora, os dois locais de diretório para 010 e 011 apontam para os dois novos buckets distintos. Antes da divisão, eles apontavam para o mesmo bucket. A profundidade local d' dos dois novos buckets é 3, que é um a mais que a profundidade local do bucket antigo.

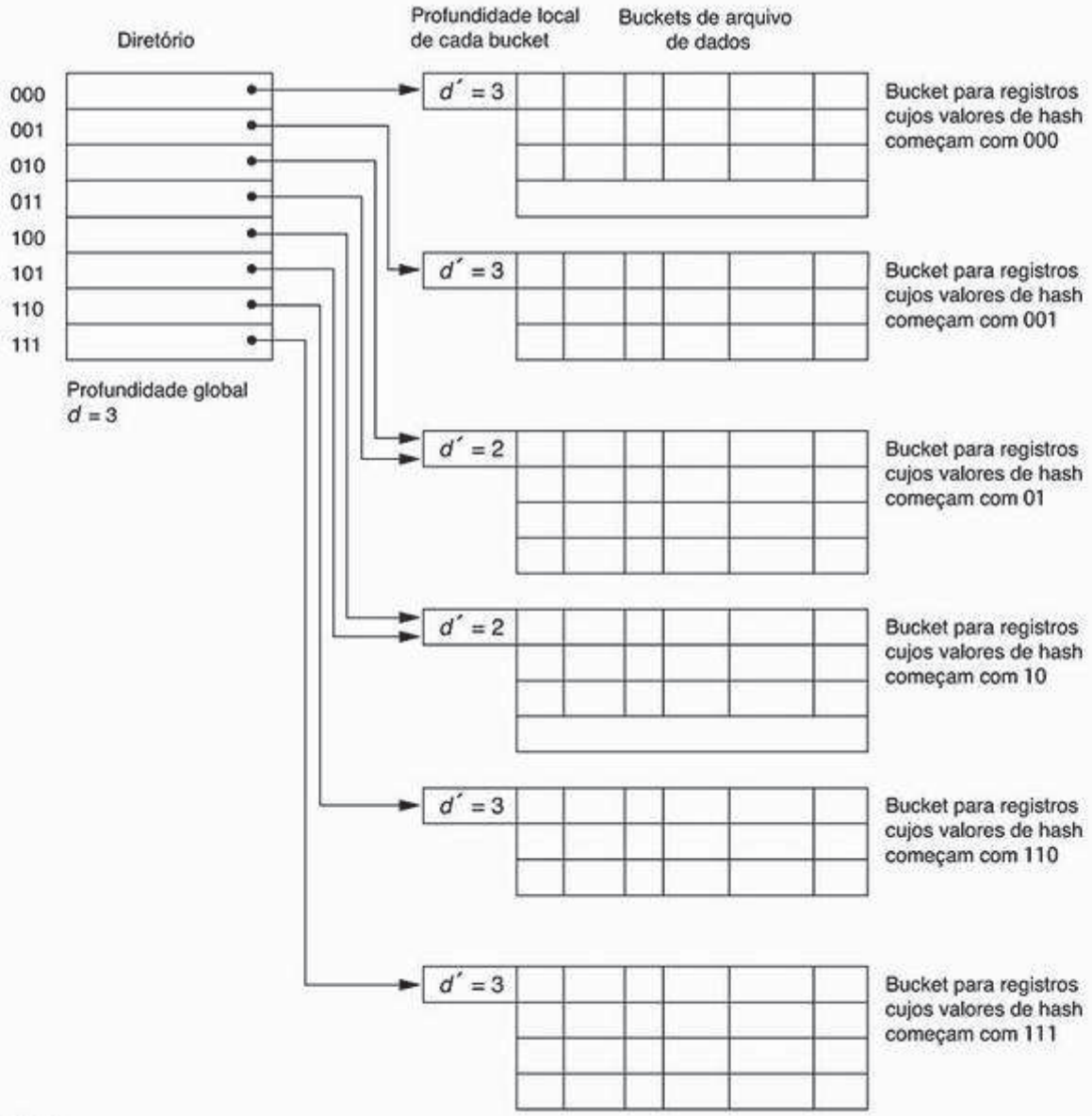

Figura 17.11
Estrutura do esquema de hashing extensível.

Snow666

Se um bucket que estoura e é dividido costumava ter uma profundidade local d' igual à profundidade global d do diretório, então o tamanho do diretório agora precisa ser dobrado de modo que possamos usar um bit extra para distinguir os dois novos buckets. Por exemplo, se o bucket para registros cujos valores de hash começam com 111 na Figura 17.11 estourar, os dois novos buckets precisam de um diretório com profundidade global $d = 4$, pois os dois buckets agora são rotulados com 1110 e 1111, e, portanto, suas profundidades locais são ambas 4. O tamanho do diretório, então, é dobrado, e cada um dos outros locais originais no diretório também é dividido em dois locais, ambos com o mesmo valor de ponteiro que o local original tinha.

A principal vantagem do hashing extensível que o torna atraente é que o desempenho do arquivo não degrada enquanto o arquivo cresce, ao contrário do hashing externo estático, onde as colisões aumentam e o encadeamento correspondente efetivamente aumenta o número médio de acessos por chave. Além disso, nenhum espaço é alocado no hashing extensível para crescimento futuro, mas buckets adicionais podem ser alocados de maneira dinâmica conforme a necessidade. O overhead de espaço para a tabela de diretório é insignificante. O tamanho de diretório máximo é 2^k, onde k é o número de bits no valor de hash. Outra vantagem é que a divisão causa uma pequena reorganização na maior parte dos casos, visto que apenas os registros em um bucket são redistribuídos para os dois novos buckets. A única ocasião em que a reorganização é mais dispendiosa é quando o diretório precisa ser dobrado (ou reduzido à metade). Uma desvantagem é que o diretório precisa ser pesquisado antes do acesso aos próprios buckets, resultando em dois acessos a bloco em vez de um no hashing estático. Essa penalidade no desempenho é considerada pequena e, portanto, o esquema é tido como bastante desejável para arquivos dinâmicos.

Hashing dinâmico. Um precursor do hashing extensível foi o hashing dinâmico, em que os endereços dos buckets eram os n bits de ordem alta ou $n - 1$ bits de ordem alta, dependendo do número total de chaves pertencentes ao respectivo bucket. O eventual armazenamento de registros em buckets para o hashing dinâmico é um tanto semelhante ao hashing extensível. A principal diferença está na organização do diretório. Enquanto o hashing extensível usa a noção de profundidade global (d bits de alta ordem) para o diretório plano e depois combina buckets redutíveis adjacentes em um bucket de profundidade local $d - 1$, o hashing dinâmico mantém um diretório estruturado em árvore com dois tipos de nós:

- Nós internos que têm dois ponteiros — o ponteiro esquerdo correspondente ao bit 0 (no endereço hashed) e um ponteiro direito correspondente ao bit 1.
- Nós folha — estes mantêm um ponteiro para o bucket real com registros.

Um exemplo do hashing dinâmico aparece na Figura 17.12. Quatro buckets são mostrados ('000', '001', '110' e '111') com endereços de três bits de ordem alta (correspondentes à profundidade global de 3) e dois buckets ('01' e '10') são mostrados com endereços de dois bits de ordem alta (correspondentes à profundidade local de 2). Os dois últimos são o resultado de reduzir o '010' e '011' para '01' e reduzir '100' e '101' para '10'. Observe que os nós de diretório são usados implicitamente para determinar as profundidades 'global' e 'local' dos buckets no hashing dinâmico. A procura por um registro, dado o endereço hashed, envolve atravessar a árvore de diretórios, que leva ao bucket que mantém esse registro. Fica para o leitor a tarefa de desenvolver algoritmos para inserção, exclusão e pesquisa de registros para o esquema de hashing dinâmico.

Hashing linear. A ideia por trás do hashing linear é permitir que um arquivo de hash expanda e encolha seu número de buckets dinamicamente sem precisar de um diretório. Suponha que o arquivo comece com M buckets numerados com 0, 1, ..., $M - 1$ e use mod a função de hash $h(K) = K \bmod M$; essa função de hash é chamada de **função de hash inicial** h_i. O overflow devido a colisões ainda é necessário e pode ser tratado ao manterem-se as cadeias de overflow individuais para cada bucket. Contudo, quando uma colisão leva a um registro de estouro em qualquer bucket de arquivo, o primeiro bucket no arquivo — bucket 0 — é dividido em dois buckets: o bucket original 0 e um novo bucket M ao final do arquivo. Os registros originalmente no bucket 0 são distribuídos entre os dois buckets com base em uma função de hashing diferente $h_{i+1}(K) = K \bmod 2M$. Uma propriedade-chave das duas funções de hash h_i e h_{i+1} é que quaisquer registros que receberam hash para o bucket 0 baseados em h_i terão um hash para o bucket 0 ou para o bucket M com base em h_{i+1}. Isso é necessário para que o hashing linear funcione.

À medida que mais colisões levam a registros de overflow, buckets adicionais são divididos na ordem *linear* 1, 2, 3, ... Se houver overflows suficientes, todos os buckets de arquivo originais 0, 1, ..., $M - 1$ terão sido divididos, de modo que o arquivo agora tem $2M$ em vez de M buckets, e todos os buckets usam a função de hash h_{i+1}. Logo, os registros no overflow por fim são redistribuídos em buckets regulares, usando a função h_{i+1} por meio de uma *divisão adiada*

Snow666

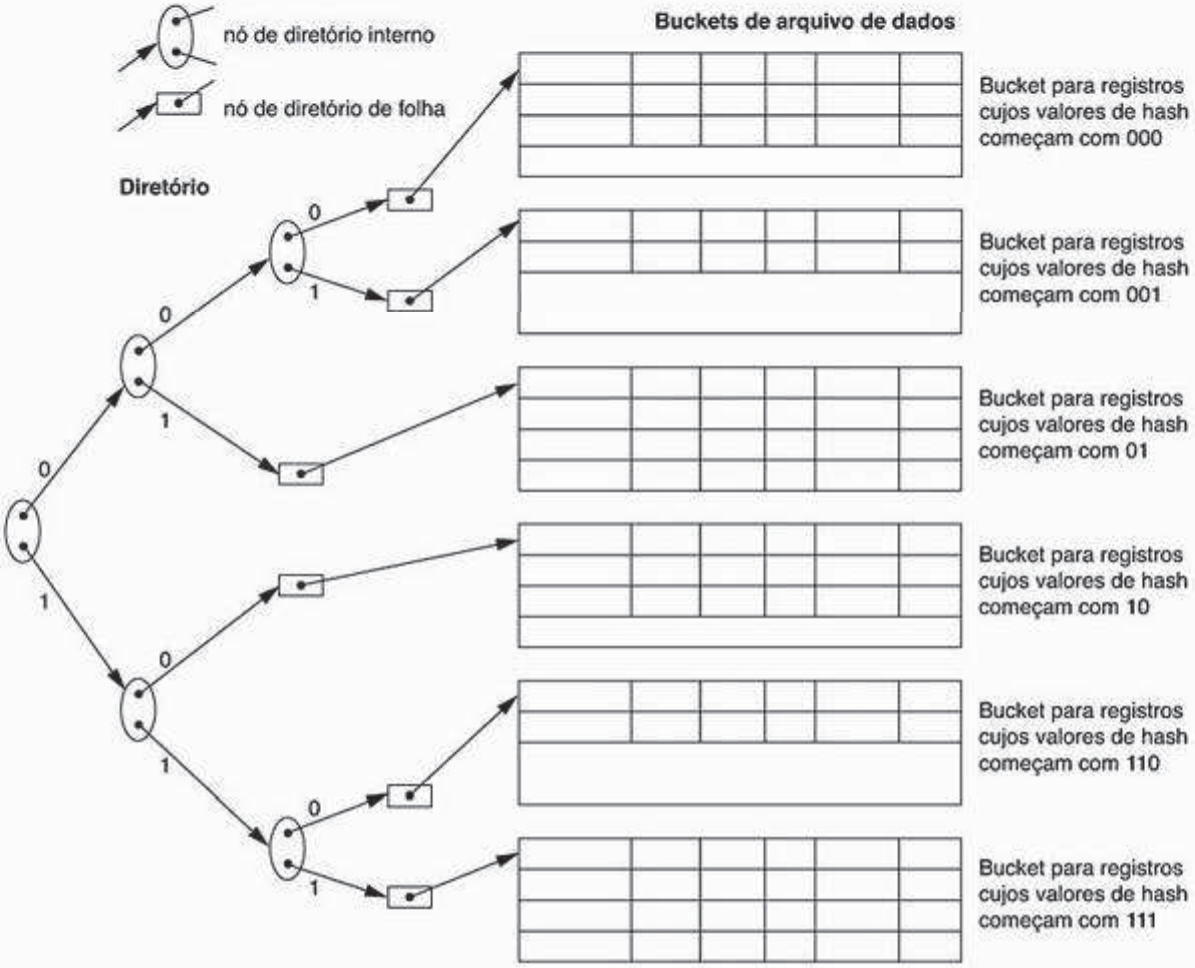

Figura 17.12
Estrutura do esquema de hashing dinâmico.

de seus buckets. Não existe diretório; somente um valor n — que é inicialmente definido como 0 e incrementado por 1 sempre que ocorre uma divisão — é necessário para determinar quais buckets foram divididos. Para recuperar um registro com valor de chave hash K, primeiro aplique a função h_i a K; se $h_i(K) < n$, então aplique a função h_{i+1} em K, porque o bucket já está dividido. Inicialmente, $n = 0$, indicando que a função h_i se aplica a todos os buckets; n cresce linearmente enquanto os buckets são divididos.

Quando $n = M$ depois de ser incrementado, isso significa que todos os buckets originais foram divididos e a função de hash h_{i+1} se aplica a todos os registros no arquivo. Nesse ponto, n é retornado a 0 (zero), e quaisquer novas colisões que causem overflow levam ao uso de uma nova função de hashing $h_{i+2}(K) = K \bmod 4M$. Em geral, uma sequência de funções de hashing $h_{i+j}(K) = K \bmod (2^jM)$ é utilizada, na qual $j = 0, 1, 2, \ldots$; uma nova função de hashing h_{i+j+1} é necessária sempre que todos os buckets 0, 1, ..., $(2^jM) - 1$ tiverem sido divididos e n for retornado a 0. A busca por um registro com valor de chave hash K é dada pelo Algoritmo 17.3.

A divisão pode ser controlada ao monitorar o fator de carga de arquivo em vez de dividir sempre que ocorre um overflow. Em geral, o **fator de carga de arquivo** l pode ser definido como $l = r/(bfr * N)$, onde r é o número atual de registros do arquivo, bfr é o número máximo de registros que podem caber em um bucket, e N é o número atual de buckets de arquivo. Os buckets que foram divididos também podem ser recombinados se o fator de carga do arquivo ficar abaixo de certo patamar. Os blocos são combinados de maneira linear, e N é reduzido adequadamente. A carga do arquivo pode ser usada para disparar divisões e combinações. Dessa maneira, ela pode ser mantida em um intervalo desejado. As divisões podem ser disparadas quando a carga excede determinado patamar — digamos, 0,9 — e as combinações podem ser disparadas quando a carga cai abaixo de

Snow666

outro patamar — digamos, 0,7. As principais vantagens do hashing linear são que ele mantém o fator de carga razoavelmente constante enquanto o arquivo aumenta e diminui, e ele não requer um diretório.[10]

Algoritmo 17.3. O procedimento de pesquisa para o hashing linear

se $n = 0$

então $m \leftarrow h_i(K)$ (* m é o valor de hash do registro com chave K *)

se não **inicio**

$m \leftarrow h_j(K)$;

se $m < n$ então $m \leftarrow h_{j+1}(K)$

fim;

procura o bucket cujo valor de hash é m (e seu overflow, se houver).

17.9 Outras organizações de arquivo primárias

17.9.1 Arquivos de registros mistos

As organizações de arquivo que estudamos até aqui consideram que todos os registros de determinado arquivo são do mesmo tipo. Os registros poderiam ser de FUNCIONARIO, PROJETO, ALUNO ou DEPARTAMENTO, mas cada arquivo contém registros de apenas um tipo. Na maioria das aplicações de banco de dados, encontramos situações em que diversos tipos de entidades são inter-relacionadas de várias maneiras, como vimos no Capítulo 7. Os relacionamentos entre registros em vários arquivos podem ser representados por **campos de conexão**.[11] Por exemplo, um registro de ALUNO pode ter um campo de conexão Dep_princ cujo valor indica o nome do DEPARTAMENTO em que o aluno está se formando. Esse campo Dep_princ *refere-se* a uma entidade DEPARTAMENTO, que deve ser representada por um registro próprio no arquivo DEPARTAMENTO. Se quisermos recuperar valores de campo de dois registros relacionados, temos de recuperar um dos registros primeiro. Depois, podemos usar seu valor de campo de conexão para recuperar o registro relacionado no outro arquivo. Logo, os relacionamentos são implementados por **referências de campo lógicas** entre os registros em arquivos distintos.

As organizações de arquivos em SGBDs de objeto, bem como em sistemas legados como os SGBDs hierárquicos e de rede, normalmente implementam relacionamentos entre registros como **relacionamentos físicos** realizados pela continuidade física (ou agrupamento) de registros relacionados ou por ponteiros físicos. Essas organizações de arquivo em geral atribuem uma área do disco para manter registros de mais de um tipo, de modo que registros de diferentes tipos podem ser **fisicamente agrupados** no disco. Se for esperado que um relacionamento em particular seja usado com frequência, a implementação física do relacionamento pode aumentar a eficiência do sistema na recuperação de registros relacionados. Por exemplo, se a consulta para recuperar um registro de DEPARTAMENTO e todos os registros de ALUNOs que estão se formando nesse departamento for frequente, seria desejável colocar cada registro de DEPARTAMENTO e seu cluster de registros de ALUNO continuamente no disco em um arquivo misto. O conceito de **agrupamento físico** dos tipos de objeto é utilizado nos SGBDs de objeto para armazenar objetos relacionados juntos em um arquivo misto.

Para distinguir os registros em um arquivo misto, cada registro tem — além de seus valores de campo — um campo de **tipo de registro**, que especifica esse item. Esse costuma ser o primeiro campo em cada registro e é usado pelo software do sistema para determinar o tipo de registro que ele está prestes a processar. Usando a informação do catálogo, o SGBD pode determinar os campos desse tipo de registro e seus tamanhos, a fim de interpretar os valores de dados nele.

17.9.2 B-trees e outras estruturas de dados como organização primária

Outras estruturas de dados podem ser usadas para organizações de arquivo primárias. Por exemplo, se tanto o tamanho do registro quanto o número de registros em um arquivo forem pequenos, alguns SGBDs oferecem a opção de uma estrutura de dados B-tree como organização de arquivo primária. Descreveremos as B-trees na Seção 18.3.1, quando discutiremos o uso da estrutura de dados B-tree para indexação. Em geral, qualquer estrutura de dados que possa ser adaptada às características dos dispositivos de disco pode ser utilizada como uma organização de arquivo primária para posicionamento de registro no disco. Recentemente, o armazenamento de dados baseado em coluna foi proposto como um método primário para armazenamento de relações nos bancos de dados relacionais. Vamos apresentá-lo rapidamente no Capítulo

[10] Para ver os detalhes sobre inserção e exclusão em arquivos com hashing linear, consulte Litwin (1980) e Salzberg (1988).

[11] O conceito de chaves estrangeiras no modelo de dados relacional (Capítulo 3) e as referências entre os objetos nos modelos orientados a objeto (Capítulo 11) são exemplos de campos de conexão.

Snow666

18 como um possível esquema de armazenamento alternativo para bancos de dados relacionais.

17.10 Paralelizando o acesso de disco usando tecnologia RAID

Com o crescimento exponencial no desempenho e na capacidade dos dispositivos semicondutores e memórias, microprocessadores mais rápidos, com memórias primárias cada vez maiores, estão continuamente se tornando disponíveis. Para corresponder a esse crescimento, é natural esperar que a tecnologia de armazenamento secundário também deva acompanhar a tecnologia do processador em desempenho e confiabilidade.

Um avanço importante na tecnologia de armazenamento secundário é representado pelo desenvolvimento do RAID, que originalmente significava **Redundant Array of Inexpensive Disks**. Mais recentemente, o *I* em RAID passou a significar *Independent*. A ideia do RAID recebeu um endosso muito positivo da indústria, e desenvolveu-se em um elaborado conjunto de arquiteturas RAID alternativas (RAID níveis 0 a 6). Destacamos os principais recursos da tecnologia nesta seção.

O objetivo principal do RAID é nivelar as diferentes taxas de melhoria de desempenho dos discos contra aquelas na memória e nos microprocessadores.[12] Enquanto as capacidades da RAM têm se quadruplicado a cada dois ou três anos, os *tempos de acesso* do disco estão melhorando em menos de 10 por cento ao ano, e as *taxas de transferência* do disco estão melhorando em aproximadamente 20 por cento ao ano. As *capacidades* do disco estão realmente melhorando em mais de 50 por cento ao ano, mas as melhorias em velocidade e tempo de acesso são de uma grandeza muito menor.

Existe uma segunda disparidade qualitativa entre a capacidade dos microprocessadores especiais de atender a novas aplicações envolvendo vídeo, áudio, imagem e processamento de dados espaciais (veja, nos capítulos 26 e 30, os detalhes sobre essas aplicações), com a correspondente falta de acesso rápido a conjuntos de dados grandes e compartilhados.

A solução natural é um grande array de pequenos discos independentes, que atuam como um único disco lógico de maior desempenho. Utiliza-se um conceito chamado **striping de dados**, que emprega o *paralelismo* para melhorar o desempenho do disco. O striping de dados distribui os dados transparentemente por vários discos, para que pareçam ser um único disco grande e rápido. A Figura 17.13 mostra um arquivo distribuído ou *striped* por quatro discos. O striping melhora o desempenho geral de E/S, permitindo que várias E/S sejam atendidas em paralelo, oferecendo assim altas taxas de transferência gerais. O striping de dados também consegue balancear a carga entre os discos. Além do mais, ao armazenar informações redundantes em discos com paridade ou algum outro código de correção de erro, a confiabilidade pode ser melhorada. Nas seções 17.10.1 e 17.10.2, discutimos como o RAID alcança os dois

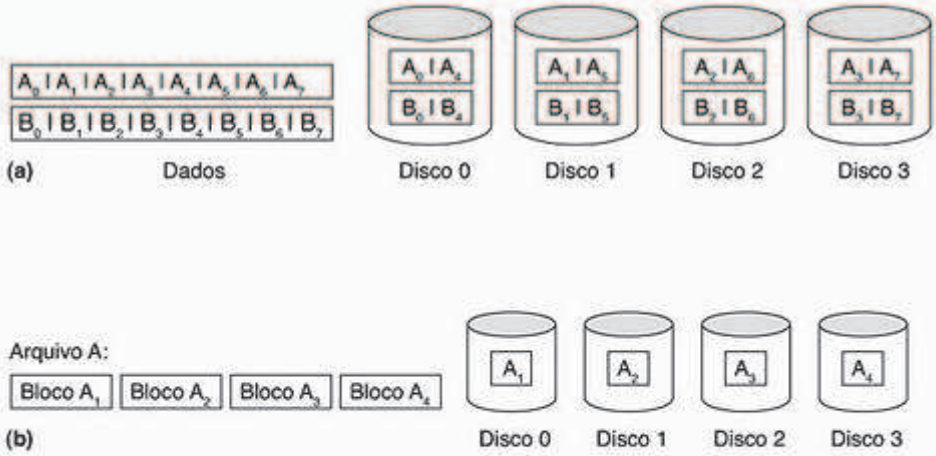

Figura 17.13
Striping de dados em vários discos. (a) Striping em nível de bit em quatro discos. (b) Striping em nível de bloco em quatro discos.

[12] Isso foi previsto por Gordon Bell para ser cerca de 40 por cento a cada ano entre 1974 e 1984, e agora deve ultrapassar os 50 por cento ao ano.

Snow666

objetivos importantes de melhor confiabilidade e maior desempenho. A Seção 17.10.3 discute as organizações e os níveis de RAID.

17.10.1 Melhorando a confiabilidade com RAID

Para um array de *n* discos, a probabilidade de falha é de *n* vezes, assim como a de um único disco. Portanto, se o MTBF (*Mean Time Between Failures*) de uma unidade de disco é considerado com 200.000 horas ou cerca de 22,8 anos (para a unidade de disco da Tabela 17.1, chamada Cheetah NS, ela é de 1,4 milhão de horas), o MTBF para um banco de 100 unidades de disco torna-se apenas 2.000 horas, ou 83,3 dias (para 1.000 discos Cheetah NS, ele seria de 1.400 horas, ou 58,33 dias). Manter uma única cópia de dados nesse tipo de array de discos causará uma perda de confiabilidade significativa. Uma solução óbvia é empregar a redundância de dados de modo que as falhas de disco possam ser toleradas. As desvantagens são muitas: operações de E/S adicionais para gravação, computação extra para manter redundância e realizar recuperação de erros, e capacidade de disco adicional para armazenar informações redundantes.

Uma técnica para introduzir redundância é chamada de **espelhamento** ou **sombreamento**. Os dados são gravados de maneira redundante em dois discos físicos idênticos, que são tratados como um disco lógico. Quando os dados são lidos, eles podem ser apanhados do disco com menores atrasos de fila, busca e rotacionais. Se um disco falhar, o outro é usado até que o primeiro seja reparado. Supondo que o tempo médio para reparo seja de 24 horas, então o tempo médio para a perda de dados de um sistema de disco espelhado que usa 100 discos com MTBF de 200.000 horas cada é $(200.000)^2/(2 * 24) = 8,33 * 10^8$ horas, que corresponde a 95.028 anos.[13] O espelhamento de disco também dobra a taxa em que as solicitações de leitura são tratadas, pois uma leitora pode ir para qualquer disco. A taxa de transferência de cada leitura, porém, permanece igual à taxa para um único disco.

Outra solução para o problema de confiabilidade é armazenar informações extras que não são necessárias normalmente, mas que podem ser usadas para reconstruir a informação pedida no caso de falha no disco. A incorporação de redundância precisa considerar dois problemas: selecionar uma técnica para calcular a informação redundante e selecionar um método de distribuição da informação redundante pelo array de disco. O primeiro problema é resolvido usando códigos de correção de erro que envolvem bits de paridade, ou códigos especializados como os códigos de Hamming. Sob o esquema de paridade, um disco redundante pode ser considerado como tendo a soma de todos os dados nos outros discos. Quando um disco falha, a informação que falta pode ser construída por um processo semelhante à subtração.

Para o segundo problema, as duas técnicas principais são armazenar a informação redundante em um pequeno número de discos ou distribuí-las uniformemente por todos os discos. A última resulta em melhor balanceamento de carga. Os diferentes níveis de RAID escolhem uma combinação dessas opções para implementar a redundância e melhorar a confiabilidade.

17.10.2 Melhorando o desempenho com RAID

Os arrays de disco empregam a técnica de striping de dados para obter taxas de transferência mais altas. Observe que os dados podem ser lidos ou gravados em apenas um bloco de cada vez, de modo que uma transferência típica contém de 512 a 8.192 bytes. O striping de disco pode ser aplicado em uma granularidade mais fina dividindo um byte de dados em bits e espalhando os bits em diferentes discos. Assim, o **striping de dados em nível de bit** consiste em dividir um byte de dados e gravar o bit *j* no *j*-ésimo disco. Com bytes de 8 bits, oito discos físicos podem ser considerados um disco lógico, com um aumento de oito vezes na taxa de transferência de dados. Cada disco participa em cada solicitação de E/S e a quantidade total de dados lidos por solicitação é oito vezes maior. O striping em nível de bit pode ser generalizado para um número de discos que é ou um múltiplo ou um fator de oito. Assim, em um array de quatro discos, o bit *n* vai para o disco que é (*n* mod 4). A Figura 17.13(a) mostra o striping de dados em nível de bit.

A granularidade da intercalação de dados pode ser mais alta do que um bit. Por exemplo, os blocos de um arquivo podem ser espalhados pelos discos, fazendo surgir o **striping em nível de bloco**. A Figura 17.13(b) mostra o striping de dados em nível de bloco considerando que o arquivo de dados contém quatro blocos. Com o striping em nível de bloco, várias solicitações independentes que acessam blocos isolados (pequenas solicitações) podem ser atendidas em paralelo por discos separados, diminuindo assim o tempo de enfileiramento das solicitações de E/S. As solicitações que acessam múltiplos blocos (grandes solicitações) podem ser feitas em paralelo, reduzindo assim o tempo de resposta.

[13] As fórmulas para cálculo do MTBF podem ser vistas em Chen et al. (1994).

Snow666

Em geral, quanto maior o número de discos em um array, maior o benefício do desempenho em potencial. Porém, considerando falhas independentes, o array de disco de 100 discos coletivamente tem 1/100 da confiabilidade de um único disco. Portanto, a redundância por meio de códigos de correção de erro e espelhamento de disco é necessária para fornecer confiabilidade junto com um desempenho alto.

17.10.3 Organizações e níveis de RAID

Diferentes organizações de RAID foram definidas com base em diversas combinações dos dois fatores de detalhamento da intercalação (striping) e do padrão de dados usados para calcular informações redundantes. Na proposta inicial, os níveis de 1 a 5 de RAID foram propostos, e dois níveis adicionais — 0 e 6 — foram acrescentados depois.

O RAID nível 0 usa striping de dados, não tem dados redundantes e, portanto, tem o melhor desempenho de gravação, pois as atualizações não precisam ser duplicadas. Ele divide os dados uniformemente entre dois ou mais discos. Porém, seu desempenho de leitura não é tão bom quanto o do RAID nível 1, que usa discos espelhados. Neste, a melhoria do desempenho é possível pelo escalonamento de uma solicitação de leitura ao disco com o menor atraso esperado de busca e rotacional. O RAID nível 2 usa a redundância no estilo da memória ao empregar códigos de Hamming, que contêm bits de paridade para subconjuntos sobrepostos distintos de componentes. Assim, em uma versão em particular desse nível, três discos redundantes são suficientes para quatro discos originais, enquanto com espelhamento — como no nível 1 —, quatro seriam necessários. O nível 2 inclui detecção e correção de erro, embora a detecção geralmente não seja exigida, pois discos defeituosos se identificam.

O RAID nível 3 utiliza um único disco de paridade contando com o controlador de disco para descobrir qual disco falhou. Os níveis 4 e 5 usam o striping de dados em nível de bloco, com o nível 5 distribuindo informações de dados e paridade por todos os discos. A Figura 17.14(b) mostra uma ilustração de RAID nível 5, em que a paridade aparece com o subscrito p. Se um disco falha, os dados que faltam são calculados com base na paridade disponível dos discos restantes. Finalmente, o RAID nível 6 se aplica ao chamado esquema de redundância $P + Q$ usando códigos de Reed-Soloman para proteger contra até duas falhas de disco usando apenas dois discos redundantes.

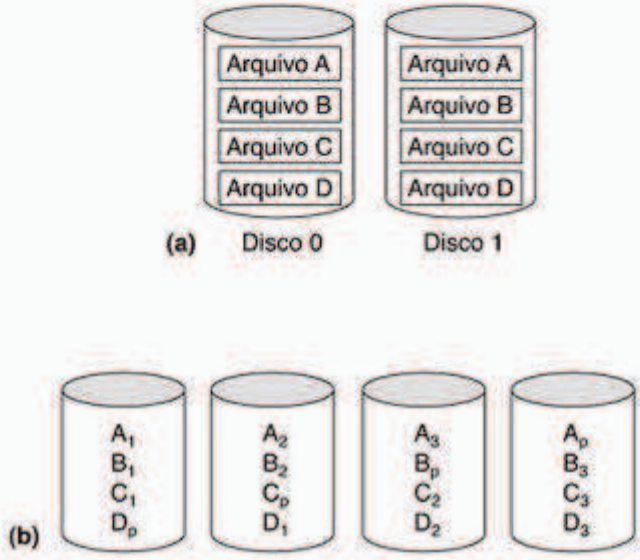

Figura 17.14
Alguns níveis de RAID populares. (a) RAID nível 1: espelhamento de dados em dois discos. (b) RAID nível 5: striping de dados com paridade distribuída por quatro discos.

A reconstrução em caso de falha de disco é mais fácil para RAID nível 1. Outros níveis exigem a reconstrução de um disco defeituoso com a leitura de múltiplos discos. O nível 1 é usado para aplicações críticas, como o armazenamento de logs de transações. Os níveis 3 e 5 são preferidos para armazenamento em grande volume, com o nível 3 oferecendo taxas de transferência mais altas. O uso mais popular da tecnologia RAID atualmente usa nível 0 (com striping), nível 1 (com espelhamento) e nível 5 com uma unidade extra para paridade. Uma combinação de vários níveis RAID também é utilizada — por exemplo, 0+1 combina striping e espelhamento usando um mínimo de quatro discos. Outros níveis de RAID fora do padrão são: RAID 1,5, RAID 7, RAID-DP, RAID S ou Parity RAID, Matrix RAID, RAID-K, RAID-Z, RAIDn, Linux MD RAID 10, IBM ServeRAID 1E e unRAID. Uma discussão sobre esses níveis fora do padrão não está no escopo deste livro. Os projetistas de uma configuração RAID para determinada mistura de aplicações precisam confrontar muitas decisões de projeto, como o nível de RAID, o número de discos, a escolha de esquemas de paridade e o agrupamento de discos para o striping em nível de bloco. Estudos de desempenho detalhados sobre leituras e gravações (referindo-se a solicitações de E/S para uma unidade de striping) e grandes leituras e gravações (referindo-se a solicitações de E/S para uma unidade de stripe de cada disco em um grupo de correção de erro) têm sido realizados.

Sn❄w66

17.11 Novos sistemas de armazenamento

Nesta seção, descrevemos três desenvolvimentos recentes em sistemas de armazenamento que estão se tornando parte integrante da maioria das arquiteturas dos sistemas de informação nas empresas.

17.11.1 Redes de área de armazenamento

Com o rápido crescimento do comércio eletrônico, sistemas de **Planejamento de Recursos Empresariais (ERP — Enterprise Resource Planning)** que integram dados da aplicação entre as organizações, e data warehouses (armazéns de dados) que mantêm informações históricas agregadas (ver Capítulo 29), a demanda por armazenamento tem crescido substancialmente. Para as organizações voltadas para a Internet de hoje, torna-se necessário passar de uma operação estática, orientada a um centro de dados fixo, para uma infraestrutura mais flexível e dinâmica para seus requisitos de processamento de informação. O custo total de gerenciamento de todos os dados está crescendo tão rapidamente que, em muitos casos, o custo de gerenciar o armazenamento ligado ao servidor ultrapassa o custo do próprio servidor. Além do mais, o custo de aquisição de armazenamento é apenas uma pequena fração — em geral, apenas 10 a 15 por cento do custo geral do gerenciamento do armazenamento. Muitos usuários de sistemas RAID não podem usar a capacidade de modo eficaz porque ela precisa estar ligada de uma maneira fixa a um ou mais servidores. Portanto, a maioria das grandes organizações mudou para um conceito chamado **áreas de armazenamento em rede (SANs — Storage Area Networks)**. Em uma SAN, os periféricos de armazenamento on-line são configurados como nós em uma rede de alta velocidade e podem ser conectados e desconectados dos servidores de uma maneira bastante flexível. Várias empresas têm surgido como provedores de SAN e fornecem as próprias topologias proprietárias. Elas permitem que os sistemas de armazenamento sejam colocados a distâncias maiores dos servidores e oferecem diferentes opções de desempenho e conectividade. As aplicações de gerenciamento de armazenamento existentes podem ser transportadas para configurações SAN por meio de redes Canal de Fibra, que encapsulam o protocolo SCSI legado. Como resultado, os dispositivos conectados à SAN aparecem como dispositivos SCSI.

As alternativas arquitetônicas atuais para SAN incluem o seguinte: conexões ponto a ponto entre servidores e sistemas de armazenamento por canal de fibra; uso de um canal de fibra para conectar vários sistemas RAID, bibliotecas de fita, e assim por diante, aos servidores; e o uso de hubs e switches de canal de fibra para conectar servidores e sistemas de armazenamento em diferentes configurações. As organizações podem lentamente passar de topologias mais simples para as mais complexas, acrescentando servidores e dispositivos de armazenamento conforme a necessidade. Não oferecemos mais detalhes aqui porque eles variam entre os vendedores de SAN. As principais vantagens alegadas são:

- Conectividade flexível de muitos-para-muitos entre servidores e dispositivos de armazenamento usando hubs e switches de canal de fibra.
- Até 10 km de separação entre um servidor e um sistema de armazenamento usando cabos de fibra ótica apropriados.
- Melhores capacidades de isolamento, permitindo o acréscimo transparente de novos periféricos e servidores.

As SANs estão crescendo muito rapidamente, mas ainda enfrentam muitos problemas, como a combinação de opções de armazenamento de vários vendedores e o tratamento dos padrões em evolução de software e hardware de gerenciamento de armazenamento. As principais empresas estão avaliando as SANs como uma opção viável para o armazenamento de banco de dados.

17.11.2 Armazenamento conectado à rede

Com o crescimento fenomenal nos dados digitais, particularmente gerados pela multimídia e outras aplicações da empresa, a necessidade de soluções de armazenamento de alto desempenho a um baixo custo tornou-se extremamente importante. Os dispositivos de **armazenamento conectado à rede (NAS — Network-Attached Storage)** estão entre os dispositivos de armazenamento usados para essa finalidade. Esses dispositivos, de fato, são servidores que não oferecem quaisquer dos serviços comuns do servidor, mas simplesmente permitem o acréscimo de armazenamento para compartilhamento de arquivos. Dispositivos NAS permitem que uma grande quantidade de espaço de armazenamento de disco rígido seja acrescentada a uma rede e podem tornar esse espaço disponível a múltiplos servidores sem ter de interrompê-los para manutenção e atualizações. Dispositivos NAS residem em qualquer lugar em uma rede local (LAN) e podem ser combinados em diferentes configurações. Um único dispositivo de hardware, normalmente chamado **caixa NAS** ou **cabeça NAS**, atua como a interface entre o sistema NAS e os clientes da rede. Esses dispositivos NAS não exigem monitor, teclado ou mouse. Uma ou mais unidades de disco ou fita podem

Snow666

ser conectadas a muitos sistemas NAS para aumentar a capacidade total. Os clientes se conectam à cabeça NAS, em vez de aos dispositivos de armazenamento individuais. Um NAS pode armazenar quaisquer dados que apareçam na forma de arquivos, como caixas de e-mail, conteúdo Web, backups de sistema remoto, e assim por diante. Nesse sentido, os dispositivos NAS estão sendo implantados como uma substituição para os servidores de arquivos tradicionais.

Os sistemas NAS trabalham por operação confiável e administração fácil. Eles incluem recursos embutidos, como autenticação segura, ou o envio automático de alertas de e-mails em caso de erro no dispositivo. Os dispositivos NAS (ou *appliances*, como alguns vendedores se referem a eles) estão sendo oferecidos com um alto grau de escalabilidade, confiabilidade, flexibilidade e desempenho. Esses dispositivos normalmente suportam RAID níveis 0, 1 e 5. As áreas de armazenamento em rede (SANs) tradicionais diferem da NAS de várias maneiras. Especificamente, as SANs costumam utilizar Canal de Fibra em vez de Ethernet, e uma SAN em geral incorpora vários dispositivos de rede ou *end points* em uma LAN autocontida ou *privativa*, enquanto a NAS conta com dispositivos individuais conectados diretamente a uma LAN pública existente. Enquanto servidores de arquivo Windows, UNIX e NetWare exigem um suporte de protocolo específico no lado do cliente, os sistemas NAS alegam ter maior independência do sistema operacional dos clientes.

17.11.3 Sistemas de armazenamento iSCSI

Um novo protocolo, chamado iSCSI (Internet SCSI) foi proposto recentemente. Ele permite que os clientes (chamados *iniciadores*) enviem comandos SCSI para dispositivos de armazenamento SCSI em canais remotos. A principal vantagem do iSCSI é que ele não exige o cabeamento especial necessário pelo Canal de Fibra e pode se estender por distâncias maiores usando a infraestrutura de rede existente. Ao transportar comandos SCSI por redes IP, o iSCSI facilita as transferências de dados pelas intranets e gerencia o armazenamento por longas distâncias. Ele pode transferir dados por redes locais (LANs), redes remotas (WANs) ou pela Internet.

O iSCSI funciona da seguinte forma. Quando um SGBD precisa acessar dados, o sistema operacional gera os comandos SCSI apropriados e a requisição de dados, que então passam por procedimentos de encapsulamento e, se for preciso, criptografia. Um cabeçalho de pacote é acrescentado antes que os pacotes IP resultantes sejam transmitidos por uma conexão Ethernet. Quando um pacote é recebido, ele é descriptografado (se foi criptografado antes da transmissão) e desmontado, separando os comandos SCSI e a solicitação. Os comandos SCSI seguem por meio do controlador SCSI para o dispositivo de armazenamento SCSI. Como o iSCSI é bidirecional, o protocolo também pode ser usado para retornar dados em resposta à solicitação original. A Cisco e a IBM comercializaram switches e roteadores com base nessa tecnologia.

O **armazenamento iSCSI** afetou principalmente empresas de pequeno e médio porte por causa de sua combinação de simplicidade, baixo custo e funcionalidade dos dispositivos iSCSI. Ele permite que elas não tenham de entender os detalhes da tecnologia Canal de fibra (FC) e, em vez disso, beneficiam-se de sua familiaridade com o protocolo IP e hardware Ethernet. As implementações iSCSI nos centros de dados de empresas muito grandes são lentas no desenvolvimento por causa de seu investimento prévio em SANs baseadas em Canal de Fibra.

O iSCSI é uma das principais técnicas de transmissão de dados de armazenamento por redes IP. O outro método, **Canal de Fibra sobre IP (CFIP)**, traduz códigos de controle Canal de Fibra e dados em pacotes IP para transmissão entre redes de área de armazenamento Canal de Fibra geograficamente distantes. Esse protocolo, também conhecido como *tunelamento Canal de Fibra* ou *tunelamento de armazenamento*, só pode ser usado junto com a tecnologia Canal de Fibra, ao passo que o iSCSI pode utilizar as redes Ethernet existentes.

A ideia mais recente a entrar na corrida do armazenamento IP da empresa é o **Canal de Fibra over Ethernet (FCoE)**, que pode ser imaginado como o iSCSI sem o IP. Ele utiliza muitos elementos de SCSI e FC (assim como o iSCSI), mas não inclui os componentes TCP/IP. Isso promete excelente desempenho, especialmente na 10 Gigabit Ethernet (10GbE), e é relativamente fácil para os vendedores acrescentarem em seus produtos.

Resumo

Começamos este capítulo discutindo as características das hierarquias de memória e depois nos concentramos nos dispositivos de armazenamento secundários. Em particular, focalizamos os discos magnéticos porque eles são usados mais frequentemente para armazenar arquivos de banco de dados on-line.

Os dados no disco são armazenados em blocos; o acesso a um bloco de disco é caro devido ao tempo de busca, atraso rotacional e tempo de transferência de bloco. Para reduzir o tempo de acesso de bloco médio, o buffering duplo pode ser usado ao acessar blocos de disco consecutivos. (Outros parâmetros de disco serão

Snow666

discutidos no Apêndice B.) Apresentamos diferentes maneiras de armazenar registros de arquivo no disco. Os registros de arquivo são agrupados em blocos de disco e podem ser de tamanho fixo ou variável, espalhados ou não espalhados, e do mesmo tipo de registro ou de tipos mistos. Discutimos sobre o cabeçalho de arquivo, que descreve os formatos de registro e mantém os endereços de disco dos blocos de arquivo. As informações no cabeçalho de arquivo são usadas pelo software do sistema que acessa os registros de arquivo.

Depois, apresentamos um conjunto de comandos típicos para acessar registros de arquivo individuais e discutimos o conceito do registro atual de um arquivo. Discutimos como as condições complexas de pesquisa de registro são transformadas em condições de pesquisa simples, utilizadas para localizar registros no arquivo.

Três organizações de arquivo primários foram então abordadas: as não ordenadas, as ordenadas e as hashed. Os arquivos desordenados exigem uma pesquisa linear para localizar registros, mas a inserção de registro é muito simples. Discutimos o problema da exclusão e o uso de marcadores de exclusão.

Os arquivos ordenados encurtam o tempo exigido para ler registros na ordem do campo de ordenação. O tempo exigido para procurar um registro qualquer, dado o valor de seu campo-chave de ordenação, também é reduzido se uma pesquisa binária for usada. Porém, manter os registros em ordem torna a inserção muito dispendiosa. Assim, a técnica de usar um arquivo de overflow desordenado para reduzir o custo de inserção de registro foi discutida. Registros de overflow são mesclados com o arquivo mestre periodicamente durante a reorganização do arquivo.

O hashing oferece acesso muito rápido a um registro qualquer de um arquivo, dado o valor de sua chave hash. O método mais adequado para o hashing externo é a técnica de bucket, com um ou mais blocos contíguos correspondendo a cada bucket. As colisões que causam overflow de bucket são tratadas pelo encadeamento. O acesso em qualquer campo não de hash é lento, e o mesmo vale para o acesso ordenado dos registros em qualquer campo. Discutimos três técnicas de hashing para arquivos que crescem e encolhem no número de registros dinamicamente: extensíveis, dinâmicos e hashing linear. Os dois primeiros usam os bits de mais alta ordem do endereço de hash para organizar um diretório. O hashing linear é preparado para manter o fator de carga do arquivo dentro de determinado intervalo e acrescentar novos buckets linearmente.

Discutimos rapidamente outras possibilidades para organizações de arquivo primárias, como as B-trees, e arquivos de registros mistos, que implementam relacionamentos entre registros de diferentes tipos fisicamente como parte da estrutura de armazenamento. Revisamos os avanços recentes em tecnologia de disco representados por RAID (Redundant Arrays of Inexpensive — ou Independent — Disks), que se tornou uma técnica-padrão em grandes empresas para oferecer melhor confiabilidade e recursos de tolerância a falhas no armazenamento. Por fim, revisamos as três opções atualmente populares nos sistemas de armazenamento empresarial: as áreas de armazenamento em rede (SANs), o armazenamento conectado à rede (NAS) e os sistemas de armazenamento iSCSI.

Perguntas de revisão

17.1. Qual é a diferença entre armazenamento primário e secundário?

17.2. Por que os discos, e não as fitas, são usados para armazenar arquivos de banco de dados on-line?

17.3. Defina os seguintes termos: *disco*, *disk pack*, *trilha*, *bloco*, *cilindro*, *setor*, *lacuna entre blocos*, *cabeça de leitura/gravação*.

17.4. Discuta o processo de inicialização de disco.

17.5. Discuta o mecanismo usado para ler ou gravar dados no disco.

17.6. Quais são os componentes de um endereço de bloco de disco?

17.7. Por que o acesso a um bloco de disco é dispendioso? Discuta os componentes de tempo envolvidos no acesso a um bloco de disco.

17.8. Como o buffering duplo melhora o tempo de acesso ao bloco?

17.9. Quais são os motivos para a existência de registros de tamanho variável? Que tipos de caracteres separadores são necessários para cada um?

17.10. Discuta as técnicas para alocar blocos de arquivo no disco.

17.11. Qual é a diferença entre uma organização de arquivo e um método de acesso?

17.12. Qual é a diferença entre arquivos estáticos e dinâmicos?

17.13. Quais são as operações típicas de um registro de cada vez para acessar um arquivo? Quais delas dependem do registro de arquivo atual?

17.14. Discuta as técnicas para exclusão de registro.

17.15. Discuta as vantagens e desvantagens do uso de (a) um arquivo desordenado, (b) um arquivo ordenado e (c) um arquivo de hash estático com buckets e encadeamento. Que operações podem ser realizadas de modo eficiente em cada uma dessas organizações, e quais operações são dispendiosas?

17.16. Discuta as técnicas para permitir que um arquivo de hash se expanda e encolha dinamicamente. Quais são as vantagens e desvantagens de cada uma?

17.17. Qual é a diferença entre os diretórios de hashing extensível e dinâmico?

17.18. Para que são usados arquivos mistos? Quais são os outros tipos de organizações de arquivo primárias?

Snow666

17.19. Descreva a divergência entre as tecnologias de processador e disco.

17.20. Quais são os principais objetivos da tecnologia RAID? Como ela os alcança?

17.21. Como o espelhamento de disco ajuda a melhorar a confiabilidade? Dê um exemplo quantitativo.

17.22. O que caracteriza os níveis na organização RAID?

17.23. Quais são os destaques dos níveis de RAID populares 0, 1 e 5?

17.24. O que são áreas de armazenamento em rede? Que flexibilidade e vantagens elas oferecem?

17.25. Descreva os principais recursos do armazenamento conectado à rede como uma solução de armazenamento empresarial.

17.26. Como os novos sistemas iSCSI melhoraram a aplicabilidade das redes da área de armazenamento?

Exercícios

17.27. Considere um disco com as seguintes características (estes não são parâmetros de qualquer unidade de disco em particular): tamanho de bloco $B = 512$ bytes; tamanho da lacuna entre blocos $G = 128$ bytes; número de blocos por trilha = 20; número de trilhas por superfície = 400. Um disk pack consiste em 15 discos de dupla face.

a. Qual é a capacidade total de uma trilha e qual é sua capacidade útil (excluindo as lacunas entre blocos)?

b. Quantos cilindros existem?

c. O que são a capacidade total e a capacidade útil de um cilindro?

d. O que são a capacidade total e a capacidade útil de um disk pack?

e. Suponha que a unidade de disco gire o disk pack a uma velocidade de 2.400 rpm (rotações por minuto); quais são a taxa de transferência (*tr*) em bytes/ms e o tempo de transferência em bloco (*btt*) em ms? Qual é o atraso rotacional (*rd*) médio em ms? Qual é a taxa de transferência em massa? (Ver Apêndice B.)

f. Suponha que o tempo de busca médio seja de 30 ms. Quanto tempo é necessário (em média) em ms para localizar e transferir um único bloco, dado seu endereço de bloco?

g. Calcule o tempo médio que seria necessário para transferir 20 blocos aleatórios e compare isso com o tempo exigido para transferir 20 blocos consecutivos usando o buffering duplo para economizar tempo de busca e atraso rotacional.

17.28. Um arquivo tem $r = 20.000$ registros de ALUNO de *tamanho fixo*. Cada registro tem os seguintes campos: Nome (30 bytes), Cpf (9 bytes), Endereco (40 bytes), TELEFONE (10 bytes), Data_nascimento (8 bytes), Sexo (1 byte), Dep_princ (4 bytes), Dep_sec (4 bytes), Tipo_aluno (4 bytes, integer) e Titulo_academico (3 bytes). Um byte adicional é usado como um marcador de exclusão. O arquivo é armazenado no disco cujos parâmetros são dados no Exercício 17.27.

a. Calcule o tamanho do registro R em bytes.

b. Calcule o fator de bloco *bfr* e o número de blocos de arquivo b, considerando uma organização não espalhada.

c. Calcule o tempo médio necessário para localizar um registro ao realizar uma pesquisa linear no arquivo se (i) os blocos do arquivo forem armazenados consecutivamente e o buffering duplo for utilizado; (ii) os blocos de arquivo não forem armazenados de maneira consecutiva.

d. Suponha que o arquivo esteja ordenado por Cpf; ao realizar uma pesquisa binária, calcule o tempo necessário para procurar um registro dado seu valor de Cpf.

17.29. Suponha que apenas 80 por cento dos registros de ALUNO do Exercício 17.28 tenham um valor para Telefone, 85 por cento para Dep_princ, 15 por cento para Dep_sec e 90 por cento para Titulo_academico; e suponha ainda que usemos um arquivo com registro de tamanho variável. Cada registro tem um *tipo de campo* de 1 byte para cada campo no registro, mais o marcador de exclusão de 1 byte e um marcador de fim de registro de 1 byte. Suponha que usemos uma organização de registro *espalhada*, em que cada bloco tem um ponteiro de 5 bytes para o próximo bloco (esse espaço não é usado para armazenamento de registro).

a. Calcule o tamanho médio do registro R em bytes.

b. Calcule o número de blocos necessários para o arquivo.

17.30. Suponha que uma unidade de disco tenha os seguintes parâmetros: tempo de busca $s = 20$ ms; atraso rotacional $rd = 10$ ms; tempo de transferência de bloco $btt = 1$ ms; tamanho de bloco $B = 2.400$ bytes; tamanho da lacuna entre blocos $G = 600$ bytes. Um arquivo FUNCIONARIO tem os seguintes campos: Cpf, 9 bytes; Ultimo_nome, 20 bytes; Primeiro_nome, 20 bytes; Minicial, 1 byte; Data_nascimento, 10 bytes; Endereco, 35 bytes; Telefone, 12 bytes; Cpf_supervisor, 9 bytes; Departamento, 4 bytes; Codigo_cargo, 4 bytes; marcador de exclusão, 1 byte. O arquivo FUNCIONARIO tem $r = 30.000$ registros, formato de tamanho fixo e blocagem não espalhada. Escreva fórmulas apropriadas *e* calcule os seguintes valores para o arquivo FUNCIONARIO acima:

Snow666

a. O tamanho do registro R (incluindo o marcador de exclusão), o fator de bloco bfr e o número de blocos de disco b.
b. Calcule o espaço desperdiçado em cada bloco de disco devido à organização não espalhada.
c. Calcule a taxa de transferência tr e a taxa de transferência em massa btf para essa unidade de disco (veja no Apêndice B as definições de tr e btr).
d. Calcule o *número de acessos de bloco* médio necessário para pesquisar um registro qualquer no arquivo, usando a pesquisa linear.
e. Calcule, em ms, o *tempo* médio necessário para pesquisar um registro qualquer no arquivo, usando a pesquisa linear, se os blocos forem armazenados em blocos de disco consecutivos e o buffering duplo for usado.
f. Calcule, em ms, o *tempo* médio necessário para pesquisar um registro qualquer no arquivo, usando a pesquisa linear, se os blocos de arquivo *não* estiverem armazenados em blocos de disco consecutivos.
g. Suponha que os registros estejam ordenados por algum campo-chave. Calcule o *número de acessos a bloco* médio e o *tempo médio* necessário para pesquisar um registro qualquer no arquivo, usando a pesquisa binária.

17.31. Um arquivo PECAS com Num_peca como chave hash inclui registros com os seguintes valores de Num_peca: 2369, 3760, 4692, 4871, 5659, 1821, 1074, 7115, 1620, 2428, 3943, 4750, 6975, 4981 e 9208. O arquivo usa oito buckets, numerados de 0 a 7. Cada bucket é um bloco de disco e mantém dois registros. Carregue esses registros no arquivo na ordem indicada, usando a função de hash $h(K) = K \bmod 8$. Calcule o número médio de acessos a bloco para uma leitura aleatória em Num_peca.

17.32. Carregue os registros do Exercício 17.31 em arquivos de hash expansíveis com base no hashing expansível. Mostre a estrutura do diretório em cada etapa, e as profundidades global e local. Use a função de hash $h(K) = K \bmod 128$.

17.33. Carregue os registros do Exercício 17.31 em um arquivo de hash expansível, usando o hashing linear. Comece com um único bloco de disco, usando a função de hash $h_0 = K \bmod 2^0$, e mostre como o arquivo cresce e como as funções de hash mudam à medida que os registros são inseridos. Suponha que os blocos sejam divididos sempre que ocorre um overflow, e mostre o valor de n em cada estágio.

17.34. Compare os comandos de arquivo listados na Seção 17.5 aos disponíveis em um método de acesso a arquivo com que você esteja acostumado.

17.35. Suponha que tenhamos um arquivo desordenado de registros de tamanho fixo que use uma organização de registro não espalhada. Esboce algoritmos para inserção, exclusão e modificação de um registro de arquivo. Informe quaisquer suposições que você fizer.

17.36. Suponha que tenhamos um arquivo ordenado de registros de tamanho fixo e um arquivo de overflow desordenado para lidar com a inserção. Os dois arquivos usam registros não espalhados. Esboce algoritmos para inserção, exclusão e modificação de um registro de arquivo e para a reorganização do arquivo. Indique quaisquer suposições que você fizer.

17.37. Você consegue pensar em técnicas que não sejam um arquivo de overflow desordenado, que possam ser usadas para tornar as inserções em um arquivo ordenado mais eficiente?

17.38. Suponha que tenhamos um arquivo de hash e registros de tamanho fixo, e suponha também que o overflow seja tratado pelo encadeamento. Esboce algoritmos para inserção, exclusão e modificação de um registro de arquivo. Indique quaisquer suposições que você fizer.

17.39. Você consegue pensar em técnicas além do encadeamento para lidar com o estouro de bucket no hashing externo?

17.40. Escreva o pseudocódigo para os algoritmos de inserção para hashing linear e para hashing extensível.

17.41. Escreva o código de programa para acessar campos individuais de registros sob cada uma das seguintes circunstâncias. Para cada caso, indique as suposições que você faz com relação a ponteiros, caracteres separadores, e assim por diante. Determine o tipo de informação necessária no cabeçalho de arquivo a fim de que seu código seja genérico em cada caso.

a. Registros de tamanho fixo com blocagem não espalhada.
b. Registros de tamanho fixo com blocagem espalhada.
c. Registros de tamanho variável com campos de tamanho variável e blocagem espalhada.
d. Registros de tamanho variável com grupos repetitivos e blocagem espalhada.
e. Registros de tamanho variável com campos opcionais e blocagem espalhada.
f. Registros de tamanho variável que permitem todos os três casos nas partes c, d e e.

17.42. Suponha que um arquivo contenha inicialmente $r = 120.000$ registros de $R = 200$ bytes cada em um arquivo desordenado (heap). O tamanho do bloco $B = 2.400$ bytes, o tempo de busca médio $s = 16$ ms, a latência rotacional

Snow666

média rd = 8,3 ms e o tempo de transferência de bloco btt = 0,8 ms. Suponha que um registro seja excluído para cada dois registros acrescentados até que o número total de registros ativos seja 240.000.

a. Quantas transferências de bloco são necessárias para reorganizar o arquivo?

b. Quanto tempo levará para encontrar um registro imediatamente antes da reorganização?

c. Quanto tempo levará para encontrar um registro imediatamente após a reorganização?

17.43. Suponha que tenhamos um arquivo sequencial (ordenado) de 10.000 registros, onde cada registro tem 240 bytes. Suponha que B = 2.400 bytes, s = 16 ms, rd = 8,3 ms e btt = 0,8 ms. Suponha que queiramos fazer X leituras de registro aleatório independentes do arquivo. Poderíamos fazer X leituras de bloco aleatórias ou poderíamos realizar uma leitura completa do arquivo inteiro procurando esses X registros. A questão é decidir quando seria mais eficiente realizar uma leitura completa do arquivo inteiro do que realizar X leituras aleatórias individuais. Ou seja, qual é o valor de X quando uma leitura completa do arquivo é mais eficiente do que X leituras aleatórias? Desenvolva isso como uma função de X.

17.44. Suponha que um arquivo de hash estático inicialmente tenha 600 buckets na área principal e que registros sejam inseridos para criar uma área de overflow de 600 buckets. Se reorganizarmos o arquivo de hash, podemos assumir que a maior parte do overflow é eliminada. Se o custo de reorganizar o arquivo é o custo das transferências de bucket (leitura e gravação de todos os buckets) e a única operação de arquivo periódica é a operação de busca, então quantas vezes teríamos de realizar uma busca (bem-sucedida) para tornar o custo da reorganização econômico? Ou seja, o custo de reorganização e o custo de pesquisa subsequente são menores que o custo de pesquisa antes da reorganização. Explique sua resposta. Considere s = 16 ms, rd = 8,3 ms e btt = 1 ms.

17.45. Suponha que queiramos criar um arquivo de hash linear com um fator de carga de arquivo de 0,7 e um fator de bloco de 20 registros por bucket, que deve conter 112.000 registros inicialmente.

a. Quantos buckets devemos alocar na área principal?

b. Qual deve ser o número de bits usados para endereços de bucket?

Bibliografia selecionada

Wiederhold (1987) possui uma discussão e análise detalhadas de dispositivos de armazenamento secundários e organizações de arquivo como uma parte do projeto de banco de dados. Os discos óticos são descritos em Berg e Roth (1989) e analisados em Ford e Christodoulakis (1991). A memória flash é discutida por Dipert e Levy (1993). Ruemmler e Wilkes (1994) apresentam um estudo da tecnologia de disco magnético. A maioria dos livros-texto sobre bancos de dados inclui discussões do material apresentado aqui. A maioria dos livros-texto de estruturas de dados, incluindo Knuth (1998), discute o hashing estático com mais detalhes; Knuth traz uma discussão completa das funções de hash e técnicas de resolução de colisão, bem como sua comparação de desempenho. Knuth também oferece uma discussão detalhada sobre as técnicas para classificação de arquivos externos. Os livros-texto sobre estruturas de arquivo incluem Claybrook (1992), Smith e Barnes (1987) e Salzberg (1988). Eles discutem organizações de arquivo adicionais, incluindo arquivos estruturados em árvore, e possuem algoritmos detalhados para operações sobre arquivos. Salzberg et al. (1990) descrevem um algoritmo de classificação externa distribuída. As organizações de arquivo com um alto grau de tolerância a falhas são descritas por Bitton e Gray (1988) e por Gray et al. (1990). O striping de disco foi proposto em Salem e Garcia Molina (1986). O primeiro artigo sobre RAID é de Patterson et al. (1988). Chen e Patterson (1990) e o excelente estudo de RAID por Chen et al. (1994) são referências adicionais. Grochowski e Hoyt (1996) discutem as tendências futuras em unidades de disco. Diversas fórmulas para a arquitetura RAID aparecem em Chen et al. (1994).

Morris (1968) é um artigo antigo sobre hashing. O hashing extensível é descrito em Fagin et al. (1979). O hashing linear é descrito por Litwin (1980). Os algoritmos para inserção e exclusão para o hashing linear são discutidos com ilustrações em Salzberg (1988). O hashing dinâmico, que apresentamos resumidamente, foi proposto por Larson (1978). Existem muitas variações propostas para o hashing extensível e linear; para ver exemplos, consulte Cesarini e Soda (1991), Du e Tong (1991) e Hachem e Berra (1992).

Os detalhes dos dispositivos de armazenamento em disco podem ser encontrados nos sites do fabricante (por exemplo, <http://www.seagate.com>, <http://www.ibm.com>, <http://www.emc.com>, <http://www.hp.com>, <http://www.storagetek.com>. A IBM tem um centro de pesquisa de tecnologia de armazenamento na IBM Almaden (<http://www.almaden.ibm.com/>).