APÊNDICE D

Estruturas de armazenamento e métodos de acesso

D.1 INTRODUÇÃO

Neste apêndice, apresentamos uma análise tutorial das técnicas normalmente usadas nos sistemas atuais para representar e acessar fisicamente o banco de dados no disco. (*Nota*: Usamos o termo *disco* o tempo todo para nos referirmos genericamente a todos os meios de acesso direto, inclusive, por exemplo, arrays RAID, armazenamento em massa, discos óticos e assim por diante, além dos próprios discos magnéticos de cabeça móvel.) Consideramos que você possui uma familiaridade básica com arquitetura de disco e entende o que significam os termos *tempo de busca*, *espera rotacional*, *cilindro*, *trilha*, *cabeça de leitura/gravação* etc. Bons tutoriais sobre esse assunto podem ser encontrados em muitos lugares; veja, por exemplo, a referência [D.4].

A questão central que motiva toda a tecnologia de estrutura de armazenamento e método de acesso é que os tempos de acesso de disco são *muito* menores do que os tempos de acesso da memória principal. Os tempos de busca e as esperas rotacionais comuns são na ordem de 5 ou 6 milissegundos e velocidades de transferência de dados típicas variam de 5 a 10 milhões de bytes por segundo; portanto, o acesso de memória principal provavelmente deve ser, no mínimo, quatro ou cinco ordens de magnitude mais rápido do que o acesso de disco em qualquer dado sistema. Um importante objetivo de desempenho é, então, **minimizar o número de acessos ao disco** (ou E/S's de disco). Este apêndice se concentra nas técnicas para atin-

gir esse objetivo – ou seja, técnicas para arranjar dados no disco de modo que quaisquer dados necessários, como algum registro específico, possam ser localizados com o mínimo de E/S possível. *Nota*: Em todo este apêndice, como nossas discussões são todas em nível de armazenamento, usamos uma terminologia em nível de armazenamento (em especial, *arquivos*, *registros* e *campos*, significando arquivos, registros e campos *armazenados*, respectivamente) em vez da terminologia relacional.

Como já foi sugerido, qualquer arranjo de dados no disco é chamado de uma **estrutura de armazenamento**. Muitas estruturas de armazenamento diferentes podem ser e ter sido criadas e, é claro, diferentes estruturas apresentam diferentes características de desempenho; algumas são apropriadas para certas aplicações, outras são boas para outras. Provavelmente, não existe uma única estrutura que seja ideal para todas as aplicações possíveis. Isso significa que um bom sistema deve suportar uma variedade de estruturas diferentes para que diversas partes do banco de dados possam ser armazenadas de diferentes maneiras e a estrutura de armazenamento para uma determinada parte possa ser alterada quando as exigências de desempenho mudarem ou se tornarem mais conhecidas.

A estrutura do apêndice é a seguinte. Depois desta seção introdutória, a Seção D.2 explica, resumidamente, o que está envolvido no processo geral de localizar e acessar um determinado registro e identifica os principais componentes de software envolvidos nesse processo. A Seção D.3, então, se aprofunda um pouco mais em dois desses componentes, o *gerenciador de arquivos* e o *gerenciador de discos*. Essas duas seções (D.2 e D.3) precisam apenas ser folheadas em uma primeira leitura; muitos detalhes que elas contêm não são realmente necessários para compreender o material subseqüente. As próximas quatro seções (D.4 a D.7), no entanto, não devem ser apenas folheadas, já que elas representam a parte mais importante de toda a discussão; elas descrevem algumas das estruturas de armazenamento mais comuns encontradas nos sistemas atuais, sob os títulos “Indexação”, “Hashing”, “Cadeias de ponteiros” e “Técnicas de compactação”, respectivamente. Finalmente, a Seção D.8 apresenta um resumo e uma breve conclusão.

Nota: A ênfase do início ao fim é dada aos *conceitos*, não aos detalhes. O objetivo é explicar a idéia geral por trás de noções como indexação, hashing etc., sem aprofundar demais nos pormenores de qualquer sistema ou técnica especificamente. Se você deseja conhecer esses detalhes, veja os livros e documentos listados na seção “Referências e bibliografia”, no final deste apêndice.

D.2 ACESSO AO BANCO DE DADOS: VISÃO GERAL

Antes de entrarmos em nossa análise das estruturas de armazenamento propriamente ditas, precisamos considerar o que está envolvido no processo de acesso de dados em geral. Localizar uma determinada informação no banco de dados e apresentá-la ao usuário envolve várias camadas de software diferentes. É claro, os detalhes dessas camadas variam consideravelmente de sistema para sistema, assim como a terminologia, mas os princípios são bastante típicos e podem ser explicados superficialmente da seguinte maneira (Ver Figura D.1).

1. Primeiro, o SGBD determina que registro é necessário e pede ao **gerenciador de arquivos** para buscar o registro. (Para os objetivos desta explicação simples, consideramos que o SGBD é capaz de identificar de antemão exatamente o registro desejado. Na prática, ele talvez precise buscar um conjunto de registros e pesquisar esses registros na memória principal para encontrar o registro desejado. Em princípio, no entanto, isso significa apenas que a seqüência das etapas 1 a 3 precisa ser repetida para cada registro desse conjunto.)
2. O gerenciador de discos, por sua vez, determina que página contém o registro desejado e pede ao **gerenciador de arquivos** para buscar essa página.
3. Finalmente, o gerenciador de discos determina o local físico da página desejada no disco e emite a requisição E/S de disco necessária. *Nota*: Algumas vezes, é claro, a página requisitada já estará no buffer na memória principal como resultado de uma busca anterior, caso em que não será necessário recuperá-la novamente.

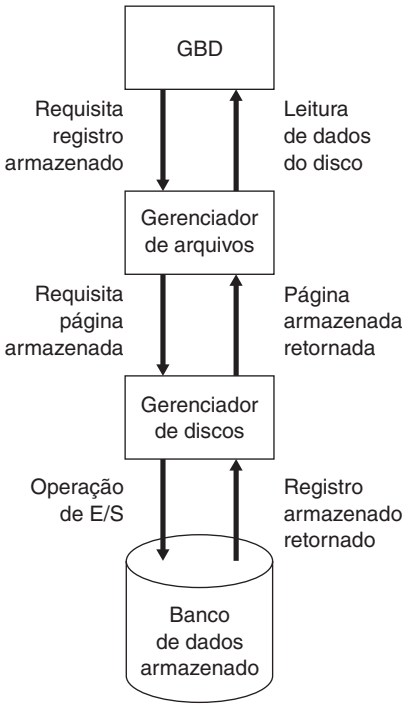

FIGURA D.1 *O SGBD, gerenciador de arquivos e gerenciador de discos*

Falando de modo geral, portanto, o SGBD possui uma visão do banco de dados como uma coleção de registros e essa visão é admitida pelo gerenciador de arquivos; o gerenciador de arquivos, por sua vez, tem uma visão do banco de dados como uma coleção de páginas e essa visão é admitida pelo gerenciador de discos; e o gerenciador de discos tem uma visão do disco "como ele realmente é". As três seções a seguir desenvolvem um pouco essas idéias e a Seção D.3 entra em mais detalhes sobre esses mesmos tópicos.

Gerenciador de discos

O gerenciador de discos é um componente do sistema operacional básico. Ele é o componente responsável por todas as operações de E/S físicas (em alguns sistemas, ele é considerado o componente "serviços de E/S básicos"). Como tal, ele certamente precisa estar ciente dos **endereços de disco físicos**. Por exemplo, quando o gerenciador de arquivos pede para buscar uma página específica *p*, o gerenciador de discos precisa saber exatamente onde a página *p* se encontra no disco. Porém, o usuário do gerenciador de discos – ou seja, o gerenciador de arquivos – *não* precisa conhecer essa informação. Em vez disso, o gerenciador de arquivos vê o disco simplesmente como uma coleção lógica de **conjuntos de páginas**, cada um deles consistindo em uma coleção de páginas de tamanho fixo. Cada conjunto de páginas é identificado por um ID de conjunto de páginas único. Cada página, por sua vez, é identificada por um **número de página** que é único dentro do disco; os conjuntos de página diferentes são separados (isto é, não possuem quaisquer páginas em comum). O mapeamento entre números de página e endereços de disco físicos é conhecido e mantido pelo gerenciador de discos. A principal vantagem desse arranjo (mas não a única) é que todo código específico do dispositivo pode ser isolado dentro de um único componente de sistema (o gerenciador de discos) e os componentes de nível mais alto – sobretudo o gerenciador de arquivos – podem, assim, ser *independentes do dispositivo*.

Como acabamos de explicar, o conjunto de páginas completo no disco é dividido em uma coleção de subconjuntos distintos chamados conjuntos de páginas. Um desses conjuntos de páginas, o conjunto **espaço livre**, serve como um reservatório de páginas disponíveis (que não estão sendo usadas no momento); todos os outros são considerados como tendo dados significantes. A alocação e desalocação de páginas de e para conjuntos de páginas é realizada pelo gerenciador de discos em resposta às requisições do gerencia-

dor de arquivos. As operações admitidas pelo gerenciador de discos nos conjuntos de páginas – ou seja, as operações que o gerenciador de arquivos é capaz de requisitar – incluem as seguintes:

- Recuperar a página *p* do conjunto de páginas *s*.
- Substituir a página *p* dentro do conjunto de páginas *s*.
- Incluir uma nova página no conjunto de páginas *s* (isto é, adquirir uma página vazia do conjunto de páginas espaço livre e retornar o novo número de página *p*).
- Remover a página *p* do conjunto de páginas *s* (isto é, retornar a página *p* para o conjunto de páginas espaço livre).

Naturalmente, as duas primeiras dessas operações são as operações básicas de E/S em nível de página de que o gerenciador de arquivos necessita. As outras duas permitem que conjuntos de páginas cresçam e diminuam conforme necessário.

Gerenciador de arquivos

O gerenciador de arquivos usa os recursos do gerenciador de discos anteriormente descritos de modo a permitir que seu usuário – o SGBD – veja o disco como uma coleção de **arquivos**. Cada conjunto de páginas conterá zero ou mais arquivos. *Nota*: O SGBD pode precisar estar ciente da existência dos conjuntos de páginas, ainda que ele não seja responsável por gerenciá-los em detalhes, por questões explicadas na próxima subseção. Em especial, o SGBD pode precisar saber quando dois arquivos compartilham o mesmo conjunto de páginas ou quando dois registros compartilham a mesma página.

Cada arquivo é identificado por um **nome de arquivo** ou **ID de arquivo**, único pelo menos dentro de seu conjunto de páginas recipiente; cada registro, por sua vez, é identificado por um **número de registro** ou **ID de registro** (RID), único pelo menos dentro do seu arquivo recipiente. (Na prática, os IDs de registro normalmente são únicos não só dentro de seu arquivo recipiente mas dentro de todo o disco, já que, em geral, eles consistem na combinação de um número de página e algum valor que seja único dentro da página. Ver Seção D.3.)

As operações admitidas pelo gerenciador de arquivos sobre os arquivos – ou seja, as operações que o SGBD é capaz de requisitar – incluem as seguintes:

- Buscar o registro *r* do arquivo *f*.
- Substituir o registro *r* do arquivo *f*.
- Incluir um novo registro no arquivo *f* e retornar o novo ID de registro *r*.
- Remover o registro *r* do arquivo *f*.
- Criar um novo arquivo *f*.
- Eliminar o arquivo *f*.

Usando essas operações elementares de gerenciamento de arquivos, o SGBD é capaz de construir e manipular as estruturas de armazenamento que são o interesse principal deste apêndice (ver Seções D.4 a D.7).

Nota: Em alguns sistemas, o gerenciador de arquivos é um componente do sistema operacional básico; em outros, ele é empacotado com o SGBD. Para nossos propósitos, a distinção não é importante. Porém, devemos observar que, embora os sistemas operacionais invariavelmente forneçam esse componente, quase sempre é o caso que o gerenciador de arquivos de finalidade geral fornecido pelo sistema operacional não é perfeitamente adequado às necessidades da "aplicação" de finalidade especial que é o SGBD. Para saber mais sobre este assunto, consulte a referência [D.44].

Cluster

Não podemos deixar essa análise geral sem mencionar brevemente o *cluster de dados*. A idéia básica por trás do cluster é tentar fazer com que registros logicamente relacionados (e, portanto, geralmente usados juntos) sejam armazenados fisicamente próximos no disco. O cluster físico de dados é um fator extremamente importante para o desempenho, como se pode ver facilmente do que segue. Suponha que o registro acessado mais recentemente seja *r1* e suponha que o próximo registro necessário seja *r2*. Imagine também que *r1* esteja armazenado na página *p1* e *r2* esteja armazenado na página *p2*. Então:

1. Se *p1* e *p2* são a mesma coisa, então, acessar *r2* não exigirá absolutamente qualquer E/S físico, pois a página *p2* desejada já estará em um buffer na memória principal.
2. Se *p1* e *p2* são diferentes mas fisicamente próximos – em especial, se forem fisicamente adjacentes – então, acessar *r2* exigirá um E/S físico (a não ser, é claro, que *p2* também esteja em um buffer na memória principal), mas o tempo de busca envolvido nesse E/S será pequeno, já que as cabeças de leitura/gravação já estão próximas à posição desejada. Especificamente, o tempo de busca será *zero* se *p1* e *p2* estiverem no mesmo cilindro.

Como um exemplo de cluster, consideramos o banco de dados comum de fornecedores e peças[1]:

- Se o acesso seqüencial a todos os fornecedores em ordem de número de fornecedor for uma necessidade freqüente da aplicação, então os registros de fornecedor devem estar clusterizados de modo que o registro F1 de fornecedor esteja fisicamente próximo do registro F2 de fornecedor, o registro F2 de fornecedor esteja fisicamente próximo do registro F3 de fornecedor e assim por diante. Este é um exemplo de cluster **intra-arquivo**: o cluster é aplicado dentro de um mesmo arquivo.
- Se, por outro lado, o acesso a algum fornecedor específico juntamente com todos os carregamentos para esse fornecedor for uma necessidade freqüente da aplicação, então os registros de fornecedor e carregamento devem estar armazenados de maneira interfoliada, com os registros de carregamento para o fornecedor F1 fisicamente próximos do registro F1 de fornecedor, os registros de carregamento para o fornecedor F2 fisicamente próximos do registro F2 de fornecedor e assim por diante. Este é um exemplo de agrupamento **interarquivo**: o cluster é aplicado através de mais de um arquivo.

É claro, um determinado arquivo ou conjunto de arquivos pode estar fisicamente clusterizado, no máximo, de uma maneira em qualquer dado momento.

O SGBD pode suportar cluster, intra-arquivo e interarquivo armazenando logicamente registros relacionados na mesma página onde for possível e em páginas adjacentes onde não for (é por isso que o SGBD pode precisar saber sobre as páginas bem como os arquivos). Quando o SGBD cria um novo registro, o gerenciador de arquivos precisa lhe permitir especificar que o novo registro seja armazenado "próximo" – ou seja, na mesma página ou, pelo menos, em uma página logicamente próxima – algum registro existente. O gerenciador de discos, por sua vez, fará o que puder para garantir que duas páginas que sejam logicamente adjacentes estejam fisicamente adjacentes no disco. Consulte a Seção D.3.

Em geral, é claro, o SGBD só poderá saber que cluster é necessário se o administrador de banco de dados for capaz de informá-lo. Um bom SGBD deve permitir que o DBA especifique diferentes tipos de cluster para diferentes arquivos. Ele também deve permitir que o cluster para um determinado arquivo ou conjunto de arquivos seja modificado se as necessidades de desempenho mudarem. Além disso, qualquer mudança no cluster físico obviamente não deve exigir alteração alguma nos programas de aplicação, se a independência de dados for desejada.

---

[1]Para simplificar, consideramos em todo este apêndice que cada fornecedor, peça ou tupla de carregamento individual é mapeado para um único registro no disco, como normalmente seria o caso na maioria dos sistemas comerciais de hoje. Veja o Apêndice A para obter mais informações.

D.3 CONJUNTOS DE PÁGINAS E ARQUIVOS

Como explicamos na seção anterior, uma importante função do gerenciador de discos é permitir que o gerenciador de arquivos ignore todos os detalhes do E/S de disco físico para pensar em termos de "E/S de página" (lógico). Essa função do gerenciador de discos é chamada de **gerenciamento de página**. Apresentaremos um exame bastante simples para mostrar como o gerenciamento de página normalmente é realizado.

Considere o banco de dados peças e fornecedores novamente. Suponha que a ordem de registros lógica desejada seja (vagamente) *seqüência de chave primária* – isto é, os fornecedores devem estar na ordem de número de fornecedor, as peças na ordem de número de peça e os carregamentos na ordem de número de peça dentro da ordem de número de fornecedor.[2] Para manter as coisas simples, suponha também que cada arquivo seja armazenado em um conjunto de páginas próprio e que cada registro exige uma página inteira própria. Agora, considere a seguinte seqüência de eventos.

1. Inicialmente, o banco de dados não contém quaisquer dados. Existe apenas um conjunto de páginas, o conjunto de páginas espaço livre, que contém todas as páginas no disco – exceto a página zero, que é especial (veja mais tarde). As outras páginas são numeradas seqüencialmente a partir de um.
2. O gerenciador de arquivos requisita a criação de um conjunto de páginas para registros de fornecedor e insere os cinco registros de fornecedor para os fornecedores F1 a F5. O gerenciador de discos remove as páginas 1 a 5 do conjunto de páginas fee-space e as rotula "o conjunto de páginas de fornecedores".
3. A mesma coisa para as peças e os carregamentos. Agora, existem quatro conjuntos de páginas: o conjunto de páginas de fornecedores (páginas 1 a 5), o conjunto de páginas de peças (páginas 6 a 11), o conjunto de páginas de carregamentos (páginas 12 a 23) e o conjunto de páginas espaço livre (páginas 24, 25, 26 etc.). A situação neste ponto é mostrada na Figura D.2.

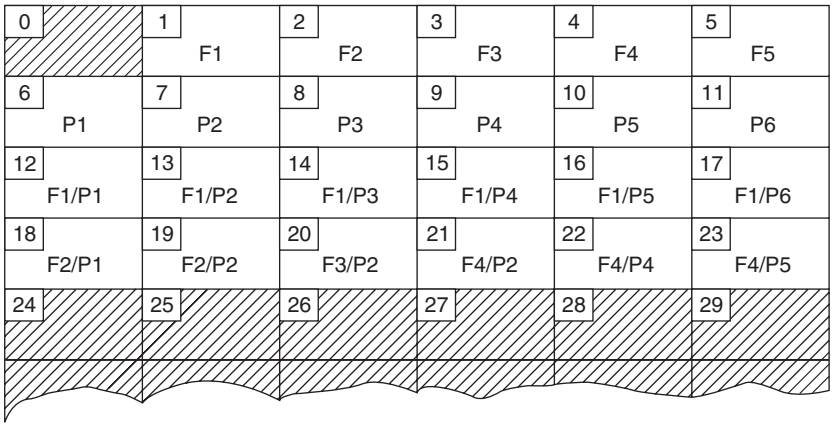

FIGURA D.2 *Layout de disco após a criação e o carregamento inicial do banco de dados peças e fornecedores*

Continuando com o exemplo:

4. A seguir, o gerenciador de arquivos insere um novo registro de fornecedor (para um novo fornecedor, o fornecedor F6). O gerenciador de discos localiza a primeira página livre no conjunto de páginas espaço livre – a página 24 – e a inclui no conjunto de páginas de fornecedores.

[2]Dizemos "vagamente" por que o termo *seqüência de chave primária* não é bem definido se a chave primária for composta. No caso dos carregamentos, por exemplo, ela pode significar qualquer ordem de número de peça dentro da ordem de número de fornecedor ou o contrário. (De qualquer modo, as chaves primárias são um conceito relacional, não um conceito em nível de arquivo; portanto, na verdade, sequer deveríamos estar falando sobre chaves primárias em nível de armazenamento.)

5. O gerenciador de arquivos exclui o registro do fornecedor F2. O gerenciador de discos retorna a página para o fornecedor F2 (página 2) ao conjunto de páginas espaço livre.
6. O gerenciador de arquivos insere um novo registro de peça (para a peça P7). O gerenciador de discos localiza a primeira página livre no conjunto de páginas espaço livre – a página 2 – e a inclui no conjunto de páginas de peças.
7. O gerenciador de arquivos exclui o registro para o fornecedor F4. O gerenciador de discos retorna a página para o fornecedor F4 (página 4) ao conjunto de páginas espaço livre.

E assim por diante. A situação nesta fase é ilustrada na Figura D.3. O ponto principal é o seguinte: Após o sistema estar rodando por algum tempo, ele não pode mais garantir que as páginas que estejam logicamente adjacentes ainda estejam fisicamente adjacentes, mesmo que elas iniciem dessa forma. Por esse motivo, a seqüência de páginas lógica em um determinado conjunto de páginas precisa ser representada não por adjacência física, mas por *ponteiros*. Cada página conterá um **cabeçalho de página** – ou seja, um conjunto de informações de controle que inclui (entre outras coisas) o endereço de disco físico da página que segue imediatamente essa página na seqüência lógica. Ver Figura D.4.

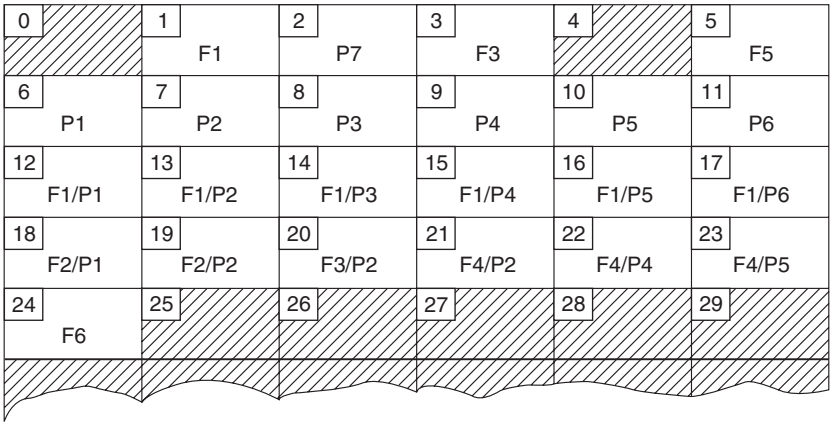

FIGURA D.3 *Layout de disco após inserir o fornecedor F6, excluir o fornecedor F2, inserir a peça P7 e excluir o fornecedor F4*

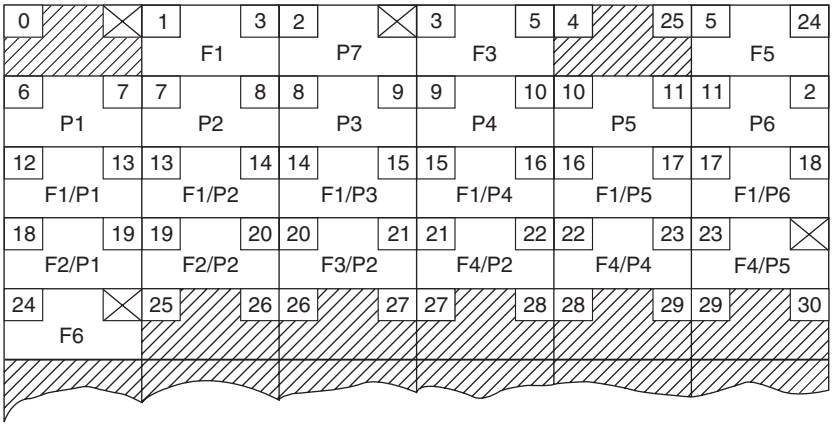

FIGURA D.4 *A Figura D.3 revisada para mostrar os ponteiros "próxima página" (canto superior direito de cada página)*

Aspectos que surgem do exemplo anterior:

- Os cabeçalhos de página – em especial, os ponteiros "próxima página" – são gerenciados pelo gerenciador de discos; eles devem estar completamente visíveis ao gerenciador de arquivos.
- Como explicado na subseção sobre cluster no final da Seção D.2, é desejável armazenar páginas logicamente adjacentes em locais fisicamente adjacentes no disco (desde que possível). Por essa razão, o gerenciador de discos normalmente aloca e desaloca páginas de e para os conjuntos de páginas, não uma de cada vez como sugerido no exemplo, mas em grupos fisicamente contíguos, ou **extensões** de, por exemplo, 64 páginas por vez.
- Surge a pergunta: Como o gerenciador de discos sabe onde os vários conjuntos de páginas estão localizados? – ou, mais precisamente, como ele sabe, para cada conjunto de páginas, onde a primeira página (logicamente) desse conjunto de páginas está localizada? (Naturalmente, basta localizar a primeira página porque a segunda página e as subseqüentes podem ser localizadas seguindo os ponteiros nos cabeçalhos de página.) A resposta é que algum local fixo no disco – normalmente, cilindro zero, trilha zero – é usado para armazenar uma página que fornece exatamente essa informação. Portanto, essa página (chamada *sumário de disco*, *diretório de disco*, *dicionário de conjunto de páginas* ou simplesmente *página zero*), em geral, contém uma lista de conjuntos de páginas existentes atualmente no disco, juntamente com um ponteiro para a primeira página de cada um desses conjuntos de páginas. Ver Figura D.5.

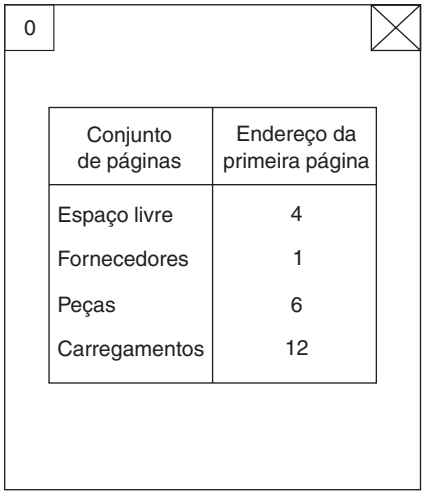

FIGURA D.5 *Diretório de disco ("página zero")*

Agora, voltemos ao gerenciador de arquivos. Assim como o gerenciador de discos permite que o gerenciador de arquivos ignore detalhes do E/S de disco físico e pense na maioria do tempo em termos de páginas lógicas, assim também o gerenciador de arquivos permite que o SGBD ignore detalhes do E/S de página e pense na maioria do tempo em termos de arquivos e registros. Essa função do gerenciador de arquivos é chamada de **gerenciamento de registro**. Discutimos essa função muito brevemente aqui, outra vez tomando o banco de dados fornecedores e peças como a base para nossos exemplos.

Suponha, então (agora, bem mais realistamente), que uma única página pode acomodar vários registros, em vez de apenas um como no exemplo de gerenciamento de página. Imagine também que a ordem lógica desejada para registros de fornecedor é a ordem de número de fornecedor, como antes. Considere a seguinte seqüência de eventos:

1. Primeiro, os cinco registros para os fornecedores F1 a F5 são inseridos e armazenados juntos em alguma página *p*, como mostra a Figura D.6. Observe que a página *p* ainda contém uma quantidade considerável de espaço livre.

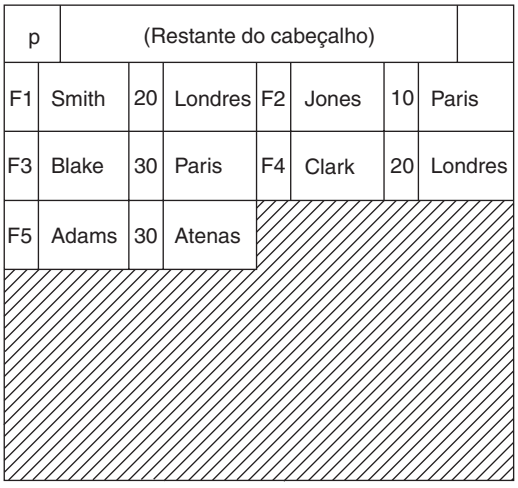

FIGURA D.6 *Layout da página p após carga inicial dos cinco registros de fornecedor para F1 e F5*

2. Agora, suponha que o SGBD insere um novo registro de fornecedor (para um novo fornecedor, digamos, fornecedor F9). O gerenciador de arquivos armazena esse registro na página *p* (porque ainda há espaço), imediatamente após o registro para o fornecedor F5.
3. O SGBD exclui o registro para o fornecedor F2. O gerenciador de arquivos exclui o registro F2 da página *p* e desloca os registros para os fornecedores F3, F4, F5 e F9 até preencher o espaço.
4. O SGBD insere um novo registro de fornecedor para outro novo fornecedor, o fornecedor F7. Novamente, o gerenciador de arquivos armazena esse registro na página *p* (porque ainda há espaço) e coloca o novo registro imediatamente após o registro para o fornecedor F5, deslocando para baixo o registro para o fornecedor F9 a fim de criar espaço. A situação nesse ponto é ilustrada na Figura D.7.

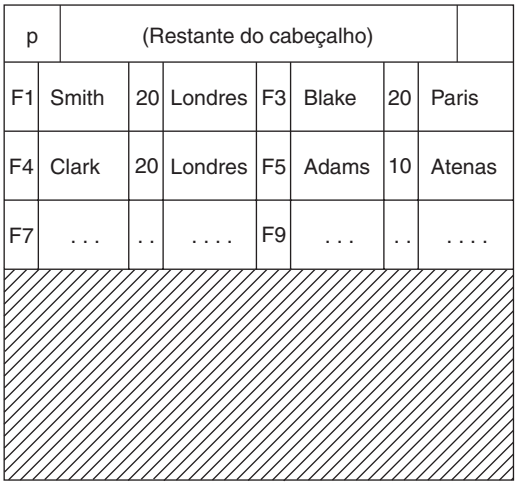

FIGURA D.7 *Layout da página p após inserir o fornecedor F9, excluir o fornecedor F2 e inserir o fornecedor F7*

E assim por diante. O principal detalhe aqui é que a seqüência lógica de registros dentro de uma determinada página pode ser representada pela seqüência *física* dentro dessa página; o gerenciador de arquivos deslocará registros para cima e para baixo para conseguir esse efeito, mantendo todos os registros de dados juntos no alto da página e todo o espaço livre junto em baixo. (É claro, a seqüência lógica de registros

através das páginas é representada pela seqüência dessas páginas dentro de seu conjunto de páginas recipiente, como descrito no exemplo de gerenciamento de página anterior.)

Como explicado na Seção D2, os registros são identificados internamente pelo *ID de registro* (RID). A Figura D.8 mostra como os RIDs normalmente são implementados. O RID para um registro *r* consiste em duas partes: (a) o número de página da página *p* contendo *r* e (b) um deslocamento de byte a partir do rodapé de *p* identificando uma vaga que contém, por sua vez, o deslocamento de byte de *r* a partir do alto de *p*. Esse esquema representa uma boa conciliação entre a velocidade do endereçamento direto e a flexibilidade do endereçamento indireto: Os registros podem ser deslocados para cima e para baixo dentro de sua página recipiente, como ilustrado nas Figuras D.7 e D.8, sem precisar mudar os RIDs (apenas os deslocamentos locais no rodapé da página precisam mudar); porém, o acesso a um determinado registro dado seu RID é rápido, envolvendo apenas um único acesso de página. (É desejável que os RIDs não mudem, pois normalmente eles são usados em qualquer outro lugar no banco de dados como ponteiros para os registros em questão – por exemplo, nos índices. Se o RID de algum registro realmente mudasse, então, todas as referências de ponteiro em outros locais precisariam mudar também.)

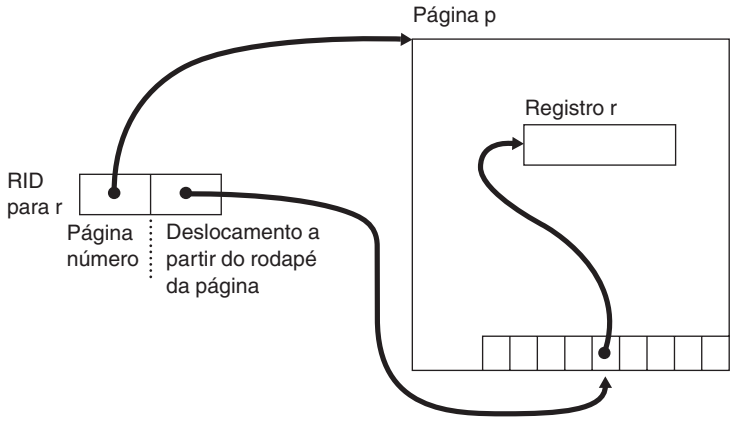

FIGURA D.8 *Implementação dos IDs de registro (RIDs)*

Nota: O acesso a um registro específico sob o esquema anterior pode, em raros casos, envolver dois acessos de página (mas nunca mais do que dois). Dois acessos serão necessários se um registro de tamanho variável for atualizado de uma maneira que se torne, agora, maior do que era antes e não haja espaço livre suficiente na página para acomodar o crescimento. Em situações como essa, o registro atualizado será colocado em outra página (uma página de **overflow**) e o registro original será, então, substituído por um ponteiro (outro RID) para o novo local. Se a mesma coisa acontecer novamente, de modo que o registro atualizado precise ser movido para uma terceira página, o ponteiro na página original será alterado para apontar para esse mais novo local.

Agora, estamos quase prontos para entrar no assunto das estruturas de armazenamento. Deste ponto em diante, iremos considerar na maior parte do tempo (exatamente como o SGBD normalmente considera) que um determinado arquivo é simplesmente uma coleção de registros, cada qual identificado de maneira única por um ID de registro que nunca muda enquanto o registro continuar existindo. Algumas questões finais para concluir esta seção:

- Note que uma conseqüência da discussão anterior é que, para qualquer arquivo, é *sempre* possível acessar seqüencialmente todos os registros desse arquivo – onde "seqüencialmente" significa "seqüência de registro dentro da seqüência de página dentro do conjunto de páginas" (em geral, aumentando a seqüência do RID). Essa seqüência costumar ser chamada grosseiramente de seqüência **física**, embora deva ficar claro que ela não corresponde necessariamente a qualquer seqüência física óbvia no disco. Por conveniência, porém, adotaremos o mesmo termo no que se segue.

- Observe que o acesso a um arquivo na seqüência física é possível mesmo se vários arquivos compartilharem o mesmo conjunto de páginas (ou seja, se os arquivos forem intercalados). Os registros que não pertencem ao arquivo em questão podem simplesmente ser saltados durante a leitura seqüencial.
- Deve ser enfatizado que a seqüência física normalmente é, no mínimo, adequada como um caminho de acesso para um determinado arquivo. Algumas vezes, ela pode até mesmo ser ideal (especialmente se o arquivo for pequeno). Porém, muitas vezes ocorre que algo melhor seja necessário. E, como descrito na Seção D.1, existe uma enorme variedade de técnicas para obter esse "algo melhor".
- Para o restante deste apêndice, por questões de simplicidade, iremos considerar normalmente que a seqüência "física" (única) para qualquer arquivo específico é a *seqüência de chave primária* (como definido na Seção D.3), salvo indicações explícitas em contrário. No entanto, veja que essa suposição é feita unicamente com a finalidade de simplificar a discussão subseqüente; reconhecemos que, na prática, pode haver boas razões para seqüenciar fisicamente um determinado arquivo de alguma outra maneira: por exemplo, pelo(s) valor(es) de algum(ns) outro(s) campo(s), ou simplesmente pelo tempo de chegada (**seqüência cronológica**).
- Por vários motivos, um registro provavelmente conterá certas informações de controle além de seus campos de dados normais. Essas informações normalmente são reunidas na frente do registro na forma de um **prefixo de registro.** Exemplos do tipo de informação encontrado nesses prefixos são o ID do arquivo recipiente (necessário se uma página puder conter registros de vários arquivos), o tamanho do registro (necessário para registros de tamanhos variáveis), um flag delete (necessário se os registros não forem excluídos fisicamente no momento de uma operação de exclusão lógica), ponteiros (necessários se os registros forem encadeados de qualquer maneira) e assim por diante. Mas, é claro, todas essas informações de controle geralmente estarão ocultas do usuário.
- Finalmente, note que os campos de dados regulares em um determinado registro serão de interesse para o SGBD *mas não para o gerenciador de arquivos* (e não para o gerenciador de discos). O SGBD precisa estar ciente desses campos porque ele os usará como base para construir índices, responder a consultas e assim por diante. O gerenciador de arquivos, no entanto, não precisa absolutamente estar ciente desses campos. Como observado no Capítulo 2, portanto, outra distinção entre o SGBD e o gerenciador de arquivos é que um determinado registro possui uma estrutura interna que é conhecida do SGBD mas não do gerenciador de arquivos (para o gerenciador de arquivos, um registro é basicamente apenas uma string de bytes).

No restante deste apêndice, descreveremos algumas das técnicas mais importantes para conseguir esse "algo melhor" (isto é, um caminho de acesso que seja melhor do que a seqüência física). As técnicas são discutidas nos títulos gerais "Indexação", "Hashing", "Cadeias de ponteiros" e "Técnicas de compactação". Uma última observação: As várias técnicas não devem ser vistas como mutuamente exclusivas. Por exemplo, é perfeitamente possível ter um arquivo, digamos, com acesso por hashing e indexado a esse arquivo baseado no mesmo campo, ou com o acesso por hashing baseado em um campo e o acesso de cadeia de ponteiros baseado em outro.

D.4 INDEXAÇÃO

Considere novamente os fornecedores. Suponha que a consulta "Obtenha todos os fornecedores na cidade *c*" (onde *c* é um parâmetro) seja uma consulta importante – isto é, uma consulta executada com freqüência e, portanto, que precisa ser bem feita. Diante dessa necessidade, o SGBD pode escolher a representação armazenada que aparece na Figura D.9. Nessa representação, há dois arquivos, um arquivo de fornecedores e um arquivo de cidades (provavelmente em conjuntos de páginas diferentes); o arquivo de cidade, que consideramos estar armazenado na seqüência de cidade (porque CIDADE é a chave primária), contém ponteiros (RIDs) para o arquivo de fornecedores. Para obter todos os fornecedores em Londres, por exemplo, o SGBD agora tem duas estratégias possíveis:

1. Pesquisar todo o arquivo de fornecedores, procurando todos os registros cujo valor "cidade" seja igual a Londres.
2. Pesquisar no arquivo de cidades as entradas Londres e, para cada uma dessas entradas, seguir o ponteiro para o registro correspondente no arquivo de fornecedores.

Se a relação dos fornecedores de Londres com os outros fornecedores for pequena, a segunda estratégia provavelmente será mais eficiente do que a primeira, já que (a) o SGBD é ciente da seqüência física do arquivo de cidades (ele pode encerrar sua pesquisa desse arquivo assim que encontrar uma cidade que venha depois de Londres na ordem alfabética) e (b), mesmo que ele precisasse pesquisar todo o arquivo de cidade, essa pesquisa provavelmente ainda exigiria menos E/S no geral, pois o arquivo de cidades é fisicamente menor do que o arquivo de fornecedores (porque os registros são menores).

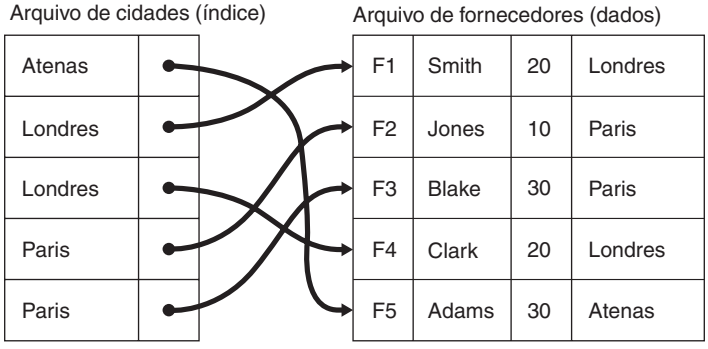

FIGURA D.9 *Indexando o arquivo de fornecedores sobre CIDADE*

Nesse exemplo, dizemos que o arquivo de cidades é um **índice** ("o índice CIDADE") para o arquivo de fornecedores; da mesma forma, dizemos que o arquivo de fornecedores é **indexado pelo** arquivo de cidades. Um índice, portanto, é um tipo especial de arquivo. Para ser específico, ele é um arquivo em que cada entrada – ou seja, cada registro – consiste exatamente em dois valores, um valor de dados e um ponteiro (RID); o valor de dados é um valor para algum campo do arquivo indexado e o ponteiro identifica um registro desse arquivo que tenha o valor para esse campo. O campo relevante do arquivo indexado é chamado de **campo indexado** ou, algumas vezes, a *chave de índice* (mas não usaremos este último termo).

Nota: Os índices são assim chamados por analogia aos índices de livro convencionais, que também consistem em entradas contendo "ponteiros" (números de página) para facilitar a busca de informações de um "arquivo indexado" (ou seja, o corpo do livro). Note, porém, que, diferente do índice CIDADE da Figura D.9, os índices de livro são *compactados hierarquicamente* – isto é, as entradas normalmente contêm vários números de página, não apenas um. Consulte a Seção D.7.

Mais terminologia: Um índice em uma chave primária – por exemplo, um índice no campo F#, no caso dos fornecedores – algumas vezes é chamado de índice *primário*. Um índice em qualquer outro campo – isto é, o índice CIDADE no exemplo – algumas vezes é chamado de índice *secundário*. Além disso, um índice em uma chave primária, ou, mais genericamente em qualquer chave candidata, costuma ser chamado de índice *único*.

Como os índices são usados

A principal vantagem de um índice é que ele torna a busca mais rápida. Mas também existe uma desvantagem: Ele torna as atualizações mais lentas. Por exemplo, toda vez que um novo registro é incluído no arquivo indexado, uma nova entrada também terá que ser incluída no índice. Como um exemplo mais específico, considere o que o SGBD precisa fazer com o índice CIDADE da Figura D.9 se o fornecedor F2 se mudar de Paris para Londres. Em geral, portanto, a pergunta que precisa ser respondida quando algum campo está sendo considerado como candidato à indexação é: O que é mais importante – uma busca efi-

ciente baseada nos valores do campo em questão ou a sobrecarga de atualização envolvida no fornecimento dessa busca eficiente?

Para o restante desta seção, iremos focalizar especificamente as operações de busca.

Os índices podem ser usados, basicamente, de duas maneiras distintas. Primeiro, eles podem ser usados para acesso **seqüencial** ao arquivo indexado – onde *seqüencial* significa "na seqüência definida pelos valores do campo indexado". Por exemplo, o índice CIDADE da Figura D.9 permitirá que os registros no arquivo de fornecedores sejam acessados na seqüência de cidade. Segundo, os índices podem ser usados para acesso **direto** a registros individuais no arquivo indexado com base em um determinado valor para o campo indexado. A consulta "Obtenha fornecedores em Londres", discutida no início da seção, ilustra este segundo caso.

Na verdade, as duas idéias básicas que acabamos de esboçar podem ser um pouco generalizadas:

- *Acesso seqüencial*: O índice também pode ajudar com consultas de *faixa* – por exemplo, "Obtenha os fornecedores cuja cidade está em alguma faixa alfabética especificada" (por exemplo, inicia com uma letra dentro da faixa L-R). Dois casos especiais importantes são (a) "Obtenha todos os fornecedores cuja cidade precede (ou segue) alfabeticamente algum valor especificado" e (b) "Obtenha todos os fornecedores cuja cidade é alfabeticamente a primeira (ou a última)".
- *Acesso direto*: O índice também pode ajudar com consultas de *lista* – por exemplo, "Obtenha todos os fornecedores cuja cidade está em alguma lista especificada (por exemplo, Londres, Paris e Nova York).

Além disso, existem certas consultas – como os *testes de existência* – que podem ser respondidas apenas através do índice, sem qualquer acesso ao arquivo indexado. Por exemplo, considere a consulta "Existe algum fornecedor em Atenas?" A resposta a essa consulta é claramente "Sim" se e somente se uma entrada para Atenas existir no índice CIDADE. Observações semelhantes se aplicam a consultas envolvendo certos operadores agregados – por exemplo, a consulta "Obtenha a primeira cidade de fornecedor na ordem alfabética", que utiliza o operador agregado NIN (essa possibilidade foi mencionada no Capítulo 18, se você se recorda).

Um determinado arquivo pode ter qualquer número de índices. Por exemplo, o arquivo de fornecedores poderia ter um índice CIDADE e também um índice STATUS (ver Figura D.10). Esses índices poderiam, então, ser usados para fornecer acesso eficiente aos registros de fornecedor com base nos valores dados para CIDADE e/*ou* STATUS. Como um exemplo do caso "ou", considere a consulta "Obtenha os fornecedores em Paris com status 30". O índice CIDADE fornece os RIDs – *r2* e *r3*, por exemplo – para os fornecedores de Paris; da mesma forma, o índice STATUS fornece os RIDs – *r3* e *r5*, por exemplo – para os fornecedores com status 30. Por esses dois conjuntos de RIDs, fica claro que o único fornecedor que satisfaz a consulta original é o fornecedor com RID igual a *r3* (a saber, o fornecedor F3). Só então o SGBD precisa acessar o arquivo de fornecedores propriamente dito para buscar o registro desejado.

FIGURA D.10 *Indexando o arquivo de fornecedores em CIDADE e STATUS*

Mais terminologia: Os índices, algumas vezes, são chamados de **listas invertidas**, pela seguinte razão. Primeiro, um arquivo "regular" – o arquivo de fornecedores das Figuras D9 e D10 pode ser tomado como um "arquivo regular" típico nesse sentido – lista, para cada registro, os valores dos campos nesse registro. Por outro lado, um índice lista, para cada valor do campo indexado, os registros que contêm esse valor. (Os sistemas de banco de dados de lista invertida mencionados brevemente no Capítulo 1, Seção 1.6, receberam seu nome dessa terminologia.) E mais um termo: Normalmente, dizemos que um arquivo com um índice em cada campo é *totalmente invertido*.

Indexando em combinações de campo

Também é possível construir um índice com base nos valores de dois ou mais campos combinados. Por exemplo, a Figura D.11 mostra um índice para o arquivo de fornecedores na combinação dos campos CIDADE e STATUS, nessa ordem. Dado esse índice, o SGBD pode responder à consulta "Obtenha fornecedores em Paris com status 30" em uma única leitura *de um único índice*. Se o índice combinado fosse substituído por dois índices separados, essa consulta envolveria duas leituras de índice separadas (como descrito anteriormente). Além disso, nesse caso, pode ser difícil decidir qual dessas duas leituras deve ser feita primeiro; como as duas seqüências possíveis podem ter características de desempenho muito diferentes, essa escolha pode ser significativa.

Observe que o índice CIDADE/STATUS combinado também pode servir como um índice unicamente no campo CIDADE, já que todas as entradas para uma determinada cidade são, pelo menos, consecutivas dentro do índice combinado. (Porém, outro índice separado terá que ser fornecido se também for necessário indexar em STATUS.) Em geral, um índice na combinação dos campos *F1*, *F2*, *F3*, ..., *Fn* (nessa ordem) também servirá como um índice apenas em *F1*, como um índice na combinação *F1F2* (ou *F2F1*), como um índice na combinação *F1F2F3* (em qualquer ordem) e assim por diante. Portanto, o número total de índices necessários para fornecer uma indexação completa dessa maneira não é tão grande quanto poderia parecer à primeira vista (veja o Exercício D.9 no final deste apêndice).

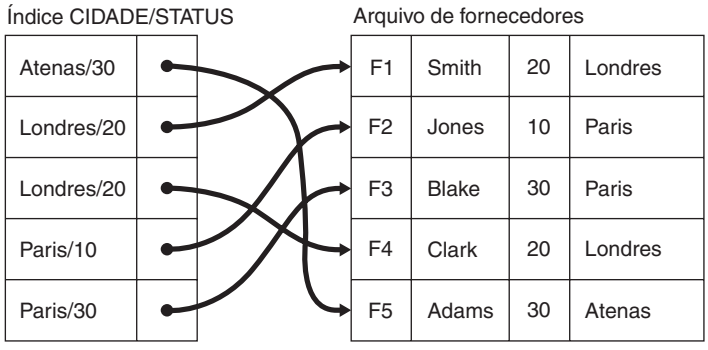

FIGURA D.11 *Indexando o arquivo de fornecedores na combinação de CIDADE e STATUS*

Indexação densa e indexação não densa

Como já dissemos várias vezes, a principal finalidade de um índice é tornar a busca mais rápida – mais especificamente, reduzir o número de E/S's de disco necessário para buscar um determinado registro. Basicamente, esse objetivo é atingido por meio dos *ponteiros*; e, até aqui, consideramos que todos os ponteiros são ponteiros de *registro* (ou seja, RIDs). Na verdade, porém, para o objetivo descrito, é suficiente que esses ponteiros sejam simplesmente ponteiros de *página* (ou seja, números de página). Para encontrar o registro desejado dentro de uma determinada página, o sistema precisará, então, realizar algum trabalho extra para pesquisar a página na memória principal, mas o número de E/S's permanecerá inalterado. *Nota*: Na realidade, a analogia de índice de livro mencionada anteriormente fornece um exemplo de índice no qual os ponteiros são ponteiros de página em vez de ponteiros de "registro".

Podemos levar essa idéia mais longe. Lembre-se de que qualquer arquivo possui uma única seqüência "física", representada pela combinação (a) da seqüência dos registros dentro de cada página e (b) da seqüência das páginas dentro do conjunto de páginas recipiente. Suponha que o arquivo de fornecedores esteja armazenado de modo que sua seqüência física corresponda à seqüência lógica como definido pelos valores de algum campo, por exemplo, o campo de número de fornecedor; em outras palavras, o arquivo de fornecedores é *Clusterizado* nesse campo (veja a discussão sobre cluster intra-arquivo no final da Seção D.2). Suponha também que um índice seja necessário nesse campo. Então, não há necessidade que o índice inclua uma entrada para cada registro no arquivo indexado (o arquivo de fornecedores, no exemplo); a única coisa necessária é uma entrada para cada *página*, fornecendo o número de fornecedor mais alto na página e o número de página correspondente. Veja a Figura D.12 (onde, para simplificação, consideramos que uma determinada página pode conter um máximo de dois registros de fornecedor).

Como um exemplo, considere o que está envolvido na busca do fornecedor F3 usando esse índice. Primeiro, o sistema precisa ler o índice, procurando a primeira entrada com número de fornecedor maior ou igual a F3. Ele encontra a entrada de índice para o fornecedor F4, que aponta para a página *p*, por exemplo. Em seguida, ele busca a página *p* e a lê na memória principal, procurando o registro requisitado (que, neste exemplo, é claro, será encontrado muito rapidamente).

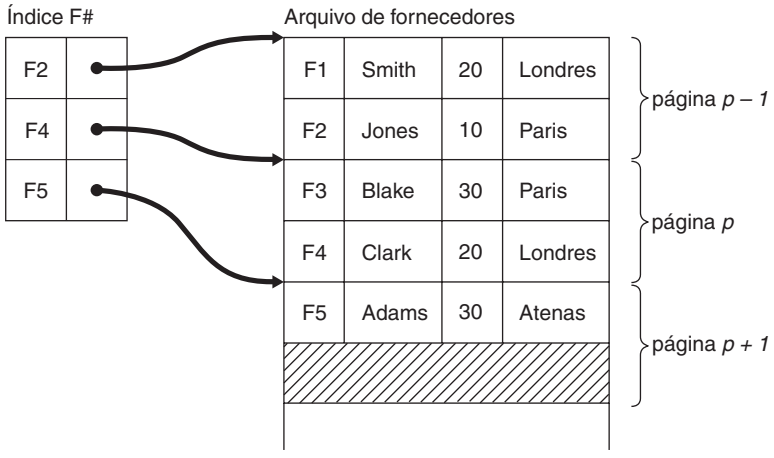

FIGURA D.12 *Exemplo de um índice não denso*

Um índice como o da Figura D.12 é chamado de **não denso** (ou, algumas vezes, esparso), pois ele não contém uma entrada para cada registro no arquivo indexado. (Por outro lado, todos os índices abordados neste apêndice até agora foram **densos**.) Uma vantagem de um índice não denso é que ele ocupará menos espaço do que um índice denso correspondente, pela razão óbvia de que ele conterá menos entradas. Como resultado, sua leitura provavelmente também será mais rápida. Uma desvantagem é que talvez não seja mais possível realizar testes de existência unicamente com base no índice (veja a breve nota sobre a realização de testes de existência na subseção "Como os índices são usados", anteriormente nesta seção).

Veja que, geralmente, um determinado arquivo pode ter, no máximo, um índice não denso, pois esse índice se baseia na seqüência física (única) do arquivo em questão. Todos os outros índices precisam necessariamente ser densos.

Árvores-b

Um tipo particularmente comum e importante de índice é a árvore-b (na verdade, a maioria dos sistemas relacionais suporta árvore-b como sua principal forma de estrutura de armazenamento, e alguns não suportam nenhuma outra forma). Porém, antes de explicarmos o que é uma árvore-b, precisamos introduzir uma outra noção preliminar: a noção de um índice **multinível** ou **estruturado em árvore**.

A razão básica para fornecer um índice é eliminar a necessidade da leitura seqüencial física do arquivo indexado. Contudo, a leitura seqüencial física ainda é necessária no *índice*. Se o arquivo indexado for muito grande, então, o índice pode se tornar bastante redimensionável e a leitura seqüencial do índice pode se tornar muito demorada. A solução para esse problema é a mesma de antes: Tratamos o índice simplesmente como um arquivo comum e construímos um índice para ele (um índice para o índice). Essa idéia pode ser estendida para tantos níveis quantos forem desejados (três são comuns na prática; um arquivo precisaria ser muito grande para exigir mais do que três níveis de indexação). Cada nível do índice age como um índice não denso para o nível abaixo (é claro, ele *precisa* ser não denso, pois, do contrário, nada seria obtido – o nível *n* conteria o mesmo número de entradas do nível $n+1$ e, portanto, levaria o mesmo tempo para ser lido).

Agora, podemos analisar as árvore-b. Uma árvore-b é um tipo especial de índice estruturado em árvore. As árvore-b, em si, foram propostos originalmente por Bayer e McCreight em 1972 [D.16]. Desde então, inúmeras variações do conceito básico foram investigados pelo próprio Bayer e por muitos outros pesquisadores; como já sugerimos, as árvore-b de um tipo ou de outro provavelmente são agora a estrutura de armazenamento mais comum em todos os sistemas de banco de dados modernos. Aqui, descrevemos a variação discutida por Knuth [D.1]. (Devemos observar, a propósito, que a estrutura de índice do IBM Virtual Storage Access Method, VSAM [D.18] é muito semelhante à estrutura de Knuth; porém, a versão VSAM foi inventada independentemente e inclui recursos adicionais próprios, como o uso de técnicas de compactação. Na verdade, um precursor da estrutura VSAM foi descrito já em 1969 [D.19].)

Na variação de Knuth, o índice consiste em duas partes, o *conjunto de seqüência* e o *conjunto de índice* (para usar a terminologia do VSAM):

- O **conjunto de seqüência** consiste em um índice de nível único para os dados reais; esse índice normalmente é denso, mas poderia ser não denso se o arquivo indexado fosse clusterizado no campo indexado. As entradas no índice, naturalmente, são agrupadas em páginas e as páginas, naturalmente, são encadeadas, de modo que a ordenação lógica representada pelo índice é obtida tomando-se as entradas na ordem física na primeira página da cadeia, seguida das entradas na ordem física da segunda página da cadeia e assim por diante. Portanto, o conjunto de seqüência fornece um acesso *seqüencial* aos dados indexados.
- O **conjunto de índice**, por sua vez, fornece acesso *direto* ao conjunto de seqüência (e, portanto, aos dados também). O conjunto de índice é, na realidade, um índice estruturado em árvore para o conjunto de seqüência; de fato, é o conjunto de índice que é a verdadeira árvore-b, estritamente falando. A combinação do conjunto de índice com o conjunto de seqüência é, algumas vezes, chamada de "b^+-tree". O nível superior do conjunto de índice consiste em um único nó (ou seja, uma única página, mas, obviamente, contendo muitas entradas de índice, como todos os outros nós). Esse nó superior é chamado de **raiz**.

Um exemplo simples é mostrado na Figura D.13. Explicamos essa figura da seguinte maneira. Primeiro, os valores 6, 8, 12, ..., 97, 99 são valores do campo indexado, por exemplo, *F*. Considere o nó superior, que consiste em dois valores *F* (50 e 82) e três ponteiros (na verdade, números de página). Os registros de dados com *F* menor ou igual a 50 podem ser encontrados (posteriormente) seguindo o ponteiro esquerdo a partir desse nó; da mesma forma, os registros com *F* maior que 50 e menor ou igual a 82 podem ser encontrados seguindo o ponteiro central; e os registros com *F* maior que 82 podem ser encontrados seguindo o ponteiro direito. Os outros nós do conjunto de índice são interpretado analogamente; observe que, por exemplo, seguir o ponteiro direito a partir do primeiro nó no segundo nível nos leva a todos os registros com *F* maior que 32 *e também menor ou igual a 50* (devido ao fato de já termos seguido o ponteiro esquerdo a partir do próximo nó mais alto).

Porém, a árvore-b (isto é, o conjunto de índice) da Figura D.13 é um tanto irrealista pelas seguintes razões:

- Primeiro, os nós de uma árvore-b normalmente não contêm todos o mesmo número de valores de dados.
- Segundo, eles normalmente contêm uma certa quantidade de espaço livre.

Em geral, uma "árvore-b de ordem *n*" possui, pelo menos, *n* mas não mais do que 2*n* valores de dados em qualquer nó (e, se tiver *k* valores de dados, ele também terá *k*+1 ponteiros). Nenhum valor de dados aparece na árvore mais de uma vez. Fornecemos o algoritmo para procurar um determinado valor *V* na estrutura da Figura D.13; o algoritmo para a árvore-b geral de ordem *n* é uma generalização simples.

```
set N para o nó raiz ;
do until N é um nó do conjunto de seqüência ;
   let X, Y (X < Y) os valores de dados no nó N ;
   if V ≤ X then set N ao nó inferior esquerdo de N ;
   if X < V ≤ Y then set N ao nó inferior intermediário de N ;
   if V > Y then set N ao nó inferior direito de N ;
end ;
if V ocorre no nó N then exit /* encontrado */ ;
if V não ocorre no nó N then exit /* não encontrado */ ;
```

Um problema com as estruturas em árvore em geral é que as inserções e exclusões podem fazer com que a árvore se torne *desequilibrada*. Uma árvore é desequilibrada quando os nós de folha não estão todos no mesmo nível – isto é, quando diferentes nós de folha estão em diferentes distâncias do nó raiz. Como a pesquisa na árvore envolve um acesso ao disco para cada nó visitado, os tempos de pesquisa podem se tornar bastante imprevisíveis em uma árvore desequilibrada. *Nota*: Na prática, o nível superior do índice – provavelmente, partes de outros níveis também – geralmente é mantido na memória principal na maior parte do tempo, o que terá o efeito de reduzir a média de acessos ao disco. Mas, o princípio geral permanece válido.

A evidente vantagem das árvore-b, por outro lado, é que os algoritmos de inserção e exclusão garantem que a árvore estará sempre equilibrada. (Alguns dizem que o "b" em "árvore-b" significa "balanced" – equilibrado – por essa razão.) Consideremos rapidamente a inserção de um novo valor, por exemplo, *V*, em uma árvore-b de ordem *n*. O algoritmo serve de instrumento apenas ao conjunto de índice, uma vez que (como explicado anteriormente) é o conjunto de índice que é a própria árvore-b; uma extensão simples também é necessária para lidar com o conjunto de seqüência.

- Primeiro, o algoritmo de pesquisa é executado para localizar não o nó de conjunto de seqüência, mas o nó (por exemplo, *N*) no nível mais baixo do conjunto de índice ao qual *V* pertence logicamente. Se *N* contiver espaço livre, *V* é inserido em *N* e o processo termina.
- Caso contrário, o nó *N* (que, portanto, precisa conter 2*n* valores) é *dividido* em dois nós, *N1* e *N2*. Suponha que S seja os 2*n* valores originais mais o novo valor *V*, em sua seqüência lógica. Os *n* valores mais baixos em S são colocados no nó *N1* esquerdo, os *n* valores mais altos em S são colocados no nó *N2* direito e o valor intermediário, digamos, *W*, é promovido ao nó pai de *N*, digamos, *P*, para servir como um valor separador para os nós *N1* e *N2*. Pesquisas posteriores de um valor *U*, ao atingir o nó *P*, serão direcionadas para o nó *N1* se $U \leq W$ e para o nó *N2* se $U > W$.
- Agora, é feita uma tentativa de inserir *W* em *P* e o processo se repete.

Na pior hipótese, ocorrerá divisão em todo caminho até o alto da árvore; será criado um novo nó raiz (pai da antiga raiz, que agora terá sido dividido em dois), e a altura da árvore crescerá em um nível (mas, ainda assim, permanecerá equilibrada).

O algoritmo de exclusão obviamente será, em essência, o inverso do algoritmo de inserção que acabamos de descrever. A mudança de um valor é feita excluindo o valor antigo e inserindo o novo valor.

D.5 HASHING

Hashing – também chamado **endereçamento de hash** e, algumas vezes, confusamente, **indexação de hash** – é uma técnica para fornecer acesso *direto* rápido a um registro específico com base em determinado valor para algum campo. O campo em questão normalmente, mas não necessariamente, é a chave primária.

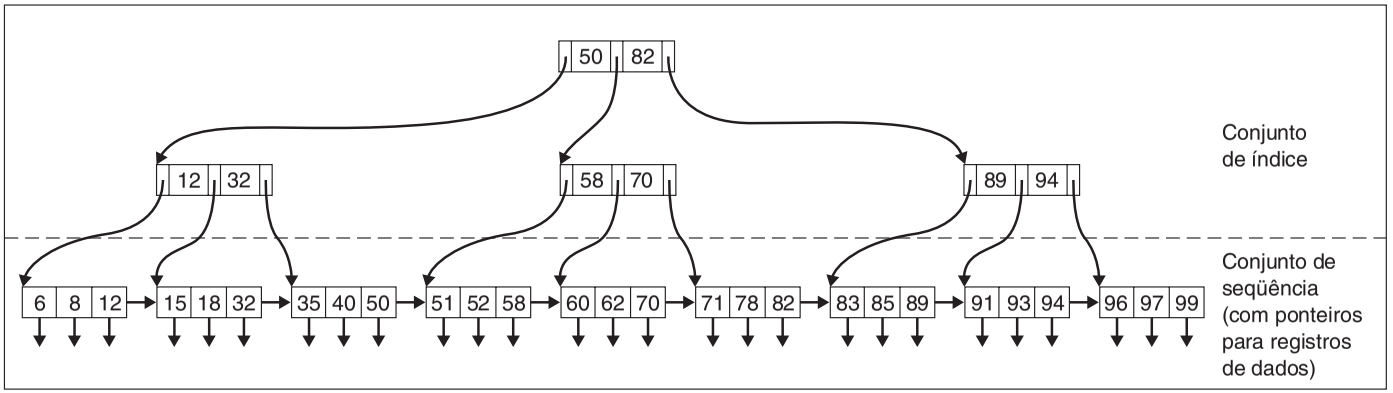

FIGURA D.13 *Parte de uma árvore-b simples (variação de Knuth)*

No geral, a técnica funciona da seguinte maneira:

- Cada registro é colocado no banco de dados em um local cujo endereço – ou seja, o RID ou, talvez, apenas o número de página – é calculado como alguma função (a **função de hash**) de algum campo desse registro (o *campo de hash*, às vezes chamado *chave de hash*; usaremos este último termo). O endereço calculado é chamado de **endereço de hash**.
- Para armazenar o registro inicialmente, o SGBD calcula o endereço de hash para o novo registro e instrui o gerenciador de arquivos a colocar o registro nessa posição.
- Para buscar o registro subseqüentemente dado o valor do campo de hash, o SGBD efetua o mesmo cálculo de antes e instrui o gerenciador de arquivos a obter o registro na posição calculada.

Como uma ilustração simples, suponha que (a) os valores de número de fornecedor sejam F100, F200, F300, F400, F500 (em vez de F1, F2, F3, F4, F5) e (b) cada registro de fornecedor requer uma página inteira para si; além disso, considere a seguinte função de hash:

```
endereço de hash (isto é, número de página) =
    resto após dividir a parte numérica do valor de F# por 13
```

Esse é um exemplo simples de uma classe bastante comum de função de hash chamada **divisão/resto**. (Por questões que fogem do objetivo deste apêndice, o divisor em um hash divisão/resto normalmente é escolhido ser um número primo, como em nosso exemplo.) Os números de página para os cinco fornecedores são, então, 9, 5, 1, 10, 6, respectivamente, produzindo a representação mostrada na Figura D.14.

Pelo que dissemos anteriormente, deve estar claro que o hashing difere da indexação, visto que, embora um determinado arquivo possa ter qualquer número de índices, ele pode ter *no máximo uma* estrutura de hash. Colocando de outra forma: Um arquivo pode ter qualquer número de campos indexados, mas apenas um campo de hash. (Essas observações consideram que o hash seja *direto*. Por outro lado, um arquivo pode ter qualquer número de hashes *indiretos*. Consulte a referência [D.24].)

Além de mostrar como o hashing funciona, o exemplo também mostra por que a função de hash é necessária. Teoricamente, seria possível usar uma função de hash "identidade" – ou seja, tomar o valor da chave primária diretamente como o endereço de hash (considerando, é claro, que a chave primária seja numérica). Porém, essa técnica geralmente seria inadequada na prática porque a faixa dos valores de chave primária possíveis normalmente seria muito mais extensa do que a faixa dos endereços disponíveis. Por exemplo, suponha que os números de fornecedor estejam realmente na faixa F000-F999, como no exemplo anterior. Então, haveria 1.000 possíveis números de fornecedor diferentes, enquanto poderia haver efetivamente apenas 10 fornecedores reais, ou algo próximo disso. Assim, para evitar que um considerável espaço de armazenamento fosse desperdiçado, gostaríamos de descobrir uma função de hash que reduzisse qualquer valor na faixa 000-999 a um valor na faixa 0-9, por exemplo. Para permitir algum espaço para crescimento, é usual estender a faixa de destino em aproximadamente 20%; é por isso que escolhemos uma função que gerasse valores na faixa 0-12 em vez de 0-9 em nosso exemplo.

O exemplo também ilustra uma das desvantagens do hashing: A "seqüência física" dos registros dentro do arquivo certamente não será a seqüência de chave primária, nem qualquer outra seqüência que tenha qualquer interpretação lógica suscetível. (Além disso, haverá lacunas de tamanho arbitrário entre registros consecutivos.) Na verdade, a seqüência física de um arquivo com uma estrutura em hashing normalmente – não invariavelmente – não é considerada para representar qualquer seqüência lógica em especial.

Nota: Evidentemente, sempre é possível *impor* qualquer seqüência lógica desejada em um arquivo em hash por meio de um índice; na verdade, é possível impor várias dessas seqüências por meio de vários índices, um para cada seqüência. Veja também as referências [D.35] e [D.37], que abordam a possibilidade de realizar hashing em esquemas que realmente preservam a seqüência lógica no arquivo quando armazenado.

Outra desvantagem do hashing em geral é que sempre existe a possibilidade de **colisões**; ou seja, dois ou mais registros diferentes ("sinônimos") que realizam hash para o mesmo endereço. Por exemplo, suponha que o arquivo de fornecedores (com fornecedores F100, F200 etc.) também inclui um fornecedor

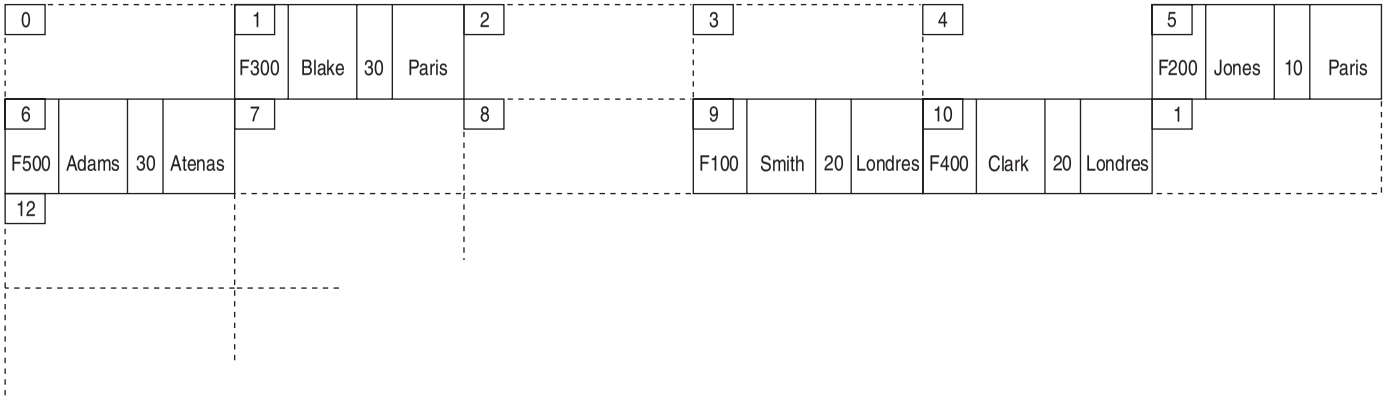

FIGURA D.14 *Exemplo de uma estrutura em hash*

com número de fornecedor F1400. Dada a função de hash em nosso exemplo ("dividir por 13 e tomar o resto"), esse fornecedor irá colidir com o fornecedor F100 no endereço de hash 9. Portanto, a função de hash da maneira que está é claramente inadequada – ela precisa ser estendida de algum modo para lidar com o problema da colisão.

Em vista do nosso exemplo original, uma extensão possível é tratar o resto após a divisão por 13, não como o endereço de hash *em si*, mas como o ponto de partida para uma leitura seqüencial. Portanto, para inserir o fornecedor F1400 (considerando que os fornecedores F100-F500 já existam), vamos para a página 9 e procuramos a primeira página livre dessa posição em diante. O novo fornecedor será armazenado na página 11. Para buscar esse fornecedor subseqüentemente, realizamos um procedimento semelhante. Esse método de **pesquisa linear** poderia ser adequado (como provavelmente é na prática) se vários registros estivessem armazenados em cada página. Suponha que cada página pode conter n registros. Então, as primeiras n colisões em algum endereço de hash p serão todas armazenadas na página p, e uma pesquisa linear nessas colisões será totalmente contida dentro dessa página. Contudo, a próxima colisão – ou seja, a $(n+1)$ésima colisão – precisará, é claro, ser armazenada em uma página de **overflow** diferente e outro E/S será necessário.

Outro método para resolver o problema da colisão, talvez mais comum nos sistemas reais, é tratar o resultado da função de hash (a, por exemplo) como o endereço de armazenamento, não para um registro de dados, mas para um **ponto de âncora**. O ponto de âncora no endereço de armazenamento a é, então, tomado como a cabeça de uma cadeia de ponteiros (uma **cadeia de colisão**) vinculando todos os registros – ou todas as páginas de registros – que colidem em a. Dentro de qualquer cadeia de colisão, as colisões normalmente serão mantidas na seqüência de campo de hash para simplificar pesquisas subseqüentes.

Hashing extensível

Ainda outra vantagem do hashing como descrito até agora é que, conforme aumenta o tamanho do arquivo em hash, o número de colisões também tende a aumentar e, portanto, o tempo de acesso médio aumenta na mesma proporção (porque cada vez mais tempo é gasto pesquisando conjuntos de colisões). Depois de algum tempo, pode-se atingir um ponto em que se torne desejável reorganizar o arquivo – ou seja, descarregar o arquivo existente e recarregá-lo, usando uma nova função de hashing.

O **hashing extensível** [D.28] é uma variação elegante do conceito básico de hashing que ameniza os problemas descritos anteriormente.[3] Na verdade, o hashing extensível garante que o número de acessos ao disco necessários para localizar um registro específico nunca será mais do que dois, normalmente sendo apenas um, por maior que possa ser o arquivo. (Portanto, isso também garante que nunca será necessária uma reorganização de arquivo.) *Nota*: Os valores do campo de hash precisam ser únicos no esquema de hashing extensível, que naturalmente serão se esse campo for realmente a chave primária, como sugerido no início desta seção.

Basicamente, o esquema funciona assim:

1. Suponha que a função de hash básica seja h e que o valor de chave primária de algum registro específico r seja k. Realizar hash em k – isto é, avaliar $h(k)$ – produz um valor s chamado pseudochave de r. As pseudochaves não são interpretadas diretamente como endereços, mas levam a locais de armazenamento de uma maneira indireta, como descrito a seguir.
2. O arquivo possui um *diretório* associado a ele, também armazenado no disco. O diretório consiste em um cabeçalho, contendo um valor d (a *profundidade* do diretório), juntamente com 2^d ponteiros. Os ponteiros apontam para páginas de dados, que contêm os registros reais (muitos registros por página). Um diretório de profundidade d, portanto, pode manipular um tamanho de arquivo máximo de 2^d páginas de dados diferentes.
3. Se considerarmos os primeiros d bits de uma pseudochave como um inteiro binário não sinalizado b, então, o i^o ponteiro no diretório ($1 \leq i \leq 2^d$) aponta para uma página que contém todos os regis-

[3] A referência [D.28] grafa "extendable" com i; portanto, "extendible" (extensível).

tros para os quais *b* toma o valor *i*-1. Em outras palavras, o primeiro ponteiro aponta para a página contendo todos os registros para os quais *b* é todos zeros, o segundo ponteiro aponta para a página para a qual *b* é 0...01 e assim por diante. (Esses 2^d ponteiros normalmente não são todos diferentes; isto é, geralmente haverá menos de 2^d páginas de dados diferentes. Ver Figura D.15.) Portanto, para encontrar o registro com valor de chave primária *k*, realizamos hash em *k* para encontrar a pseudochave *s* e tomar os primeiros *d* bits dessa pseudochave; se esses bits tiverem o valor numérico *i*-1, vamos para o *i*º ponteiro no diretório (primeiro acesso ao disco) e o seguimos até a página contendo o registro necessário (segundo acesso ao disco).

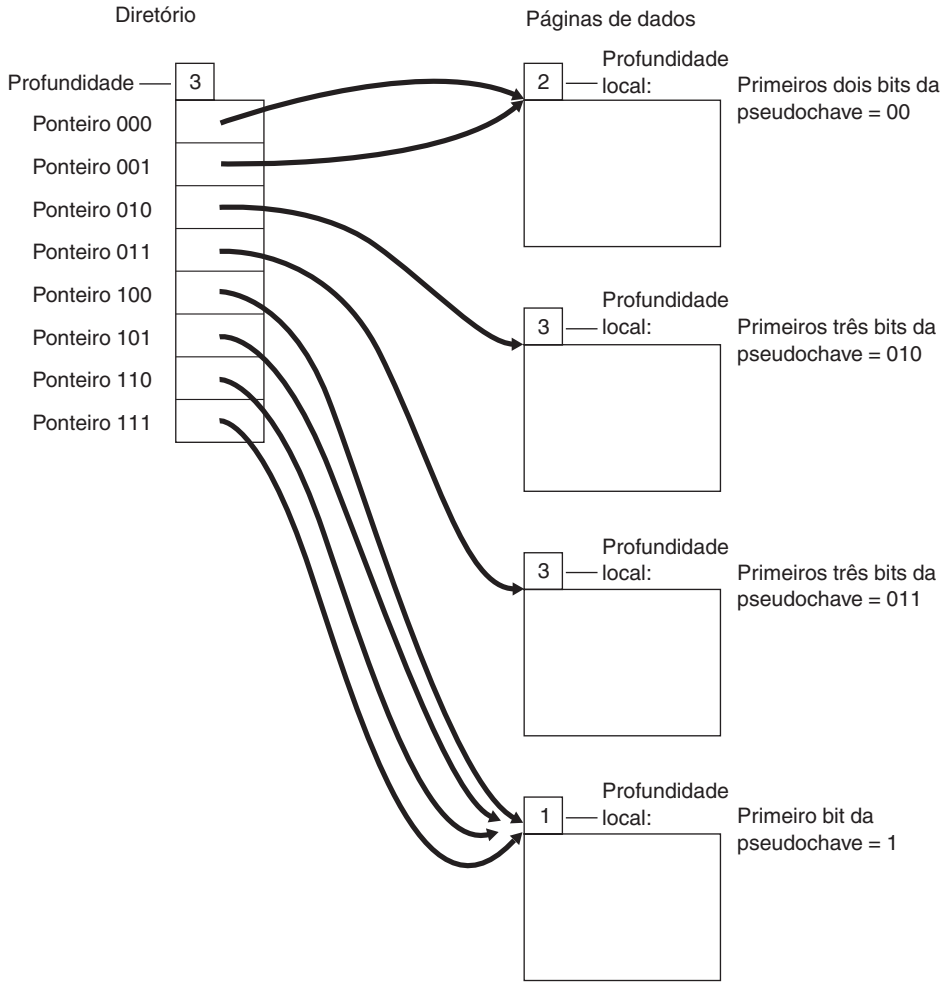

FIGURA D.15 *Exemplo de hashing extensível*

Nota: Na prática, o diretório normalmente será pequeno o bastante para que possa ser mantido na memória principal na maior parte do tempo. Os "dois" acessos ao disco, portanto, devem ser reduzidos a um, na prática.

4. Cada página de dados também possui um cabeçalho indicando a *profundidade local p* da página ($p \leq d$). Suponha, por exemplo, que *d* seja 3 e que o primeiro ponteiro no diretório (o ponteiro 000) aponte para uma página para a qual a profundidade local *p* é 2. A profundidade local 2 aqui significa que, não só essa página contém todos os registros com pseudochaves iniciando com 000, mas também contém *todos* os registros com pseudochaves iniciando com 00 (ou seja, as que iniciam com 000 e também as que iniciam com 001). Em outras palavras, o ponteiro de diretório 001 também aponta para essa página. Veja a Figura D.15 novamente.

5. Suponha, agora, que a página de dados 000 esteja cheia e queremos inserir um novo registro tendo uma pseudochave que inicia com 000 (ou 001). Nesse ponto, a página é dividida em duas: ou seja, uma nova página vazia é adquirida e todos os registros 001 são movidos da página antiga para a nova página. O ponteiro 001 no diretório é alterado para apontar para a nova página (o ponteiro 000 ainda aponta para a página antiga). A profundidade local *p* para cada uma das duas páginas, agora, será 3, não 2.
6. Agora, vamos supor que a página de dados para 000 se torna cheia novamente e precisa se dividir outra vez. O diretório existente não pode manipular essa divisão, pois a profundidade local da página a ser dividida já é igual à profundidade do diretório. Conseqüentemente, nós *duplicamos o diretório*; isto é, aumentamos *d* em 1 e substituímos cada ponteiro por um par de ponteiros adjacentes e idênticos. A página de dados, agora, pode ser dividida; os registros 0000 são deixados na página antiga e os registros 0001 vão para a nova página; o primeiro ponteiro no diretório é deixado inalterado (ou seja, ainda aponta para a página antiga) e o segundo ponteiro é modificado para apontar para a nova página. Note que dobrar o diretório é uma operação muito pouco onerosa, já que ela não envolve acesso a qualquer uma das páginas de dados.

Isso conclui o assunto do hashing extensível. Muitas outras variações sobre o conceito básico de hashing foram desenvolvidas; veja, por exemplo, as referências [D.29] a [D.36].

D.6 CADEIAS DE PONTEIROS

Suponha novamente, como no início da Seção D.4, que a consulta "Obtenha todos os fornecedores na cidade *c*" seja uma consulta importante. Outra representação armazenada que pode manipular essa consulta razoavelmente bem – talvez melhor do que um índice, embora não necessariamente assim – usa *cadeias de ponteiros.* Essa representação é ilustrada na Figura D.16. Como você pode ver, ela envolve dois arquivos, um arquivo de fornecedores e um arquivo de cidade, muito semelhante à representação de índice da Figura D.9 (desta vez, os dois arquivos provavelmente estão no mesmo conjunto de páginas, por questões a serem explicadas na Seção D.7). Porém, na representação de cadeia de ponteiros da Figura D.16, o arquivo de cidades não é um índice mas, sim, o que se chama, algumas vezes, de arquivo *pai*. O arquivo de fornecedores, de igual modo, é chamado de arquivo *filho* e a estrutura geral é um exemplo de **organização pai/filho**.

No exemplo, a estrutura pai/filho é baseada nos valores de cidade de fornecedor. O arquivo pai (cidade) contém um registro para cada cidade de fornecedor diferente, fornecendo o valor de cidade e agindo como a cabeça de uma *cadeia* ou *anel* de ponteiros vinculando todos os registros filho (fornecedores) para os fornecedores nessa cidade. Note que o campo de cidade foi removido do arquivo de fornecedores; para obter todos os fornecedores em Londres, por exemplo, o SGBD pode procurar a entrada Londres no arquivo de cidades e, depois, seguir a cadeia de ponteiros correspondente.

A principal vantagem da estrutura pai/filho é que os algoritmos de inserção e exclusão são um pouco mais simples e, conceitualmente, podem ser mais eficientes do que os algoritmos para um índice; além disso, a estrutura provavelmente ocupará menos espaço de armazenamento do que a estrutura de índice correspondente, já que cada valor de cidade aparece exatamente uma vez e não muitas vezes. As maiores desvantagens são as seguintes:

- Para uma determinada cidade, a única maneira de acessar o enésimo fornecedor é seguir a cadeia e acessar também o 1º, 2º, ..., (*n*-1)ésimo fornecedor. A menos que os registros de fornecedor sejam corretamente clusterizados, cada acesso envolverá uma operação de busca separada e o tempo gasto para acessar o enésimo fornecedor poderia ser considerável.
- Embora a estrutura possa ser adequada para a consulta "Obtenha os fornecedores em uma determinada cidade", ela não é de ajuda alguma – na verdade, ela é uma barreira positiva – para a consulta inversa "Obtenha a cidade para um determinado fornecedor" (onde o fornecedor dado é identificado por um número de fornecedor dado). Para esta última consulta, um hash ou um índice no arquivo de fornece-

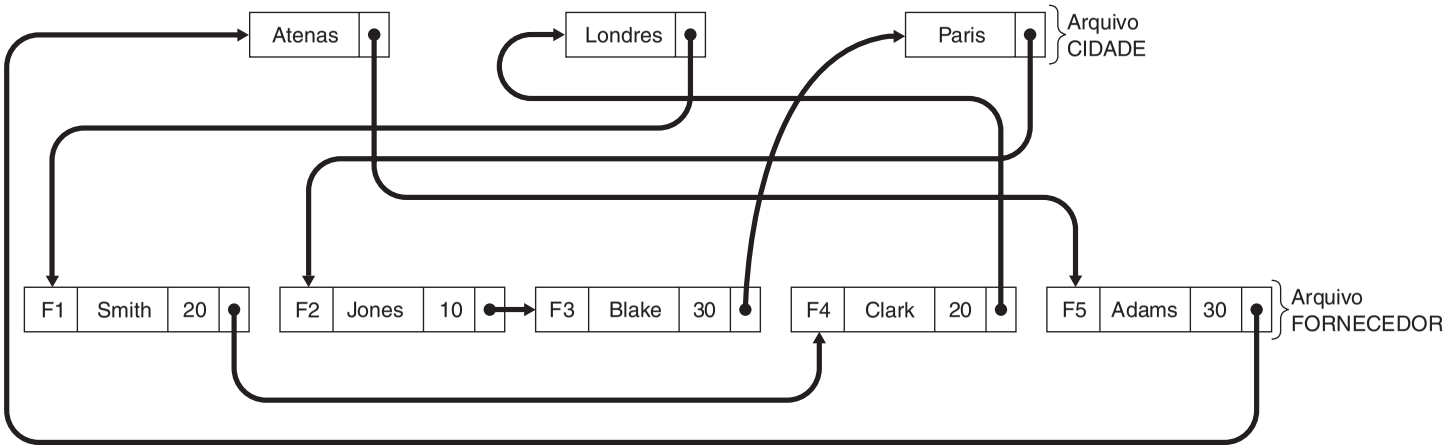

FIGURA D.16 *Exemplo de uma estrutura pai/filho*

dores provavelmente seja desejável; note que uma estrutura pai/filho baseada em números de fornecedor não faria muito sentido (por que não?). E, mesmo quando o registro de fornecedor dado é localizado, ainda é necessário seguir a cadeia até o registro pai para descobrir a cidade desejada (a necessidade dessa etapa extra é nossa justificativa para afirmar que a estrutura pai/filho é, na verdade, um obstáculo para esse tipo de consulta).

Note, ainda, que o arquivo pai (cidade) provavelmente exigirá também acesso de índice ou hash se ele for de qualquer tamanho significante. Por essa razão, as cadeias de ponteiros, sozinhas, não são realmente uma base adequada para uma estrutura de armazenamento – outros mecanismos, como os índices, certamente serão também necessários.

- Haja vista que (a) as cadeias de ponteiros, na realidade, examinam os registros – ou seja, os prefixos de registro incluem fisicamente os ponteiros relevantes – e (b) os valores do campo relevante são fatorados dos registros filho e colocados nos registros pai, a criação de uma estrutura pai/filho sobre um conjunto de registros existente constitui-se em uma difícil tarefa. Na verdade, esse tipo de operação normalmente exigirá uma reorganização do banco de dados, pelo menos para a parte relevante do banco de dados. Por outro lado, é comparativamente simples criar um novo índice sobre um conjunto de registros existente. *Nota*: Em geral, criar um novo hash também demandará uma reorganização a menos que o hash seja indireto [D.24]

Diversas variações são possíveis na estrutura pai/filho. Por exemplo:

- Os ponteiros poderiam se tornar colaborativos. Uma vantagem dessa variação é que ela simplifica o processo de ajuste de ponteiro necessário pela operação de excluir um registro filho.
- Outra extensão seria incluir um ponteiro (um "ponteiro pai") de cada registro filho diretamente ao pai correspondente. Essa extensão reduziria a quantidade de percurso de cadeia envolvido na resposta à consulta "Obtenha a cidade para um determinado fornecedor" (note, entretanto, que ela não elimina a necessidade de um hash ou índice para ajudar nessa consulta).
- Ainda outra variação seria *não* remover o campo de cidade do arquivo de fornecedores, mas repetir o campo nos registros de fornecedor (uma forma simples de redundância controlada). Certas buscas – por exemplo, "Obtenha a cidade para o fornecedor F4" – seriam, então, mais eficientes. Porém, veja que esse aumento de eficiência não tem nada a ver com a estrutura de cadeia de ponteiros em si; note também que um hash ou índice nos números de fornecedor provavelmente ainda serão necessários.

Finalmente, é claro, exatamente como é possível ter qualquer número de índices sobre um determinado arquivo, também é possível ter qualquer número de cadeias de ponteiros examinando um determinado arquivo. (Também seria possível, embora talvez incomum, ter as duas coisas.) A Figura D.17 mostra uma representação para o arquivo de fornecedores que envolve duas cadeias de ponteiros diferentes e, portanto, duas estruturas pai/filho diferentes, uma com um arquivo de cidades como pai (como na Figura D.16) e uma com um arquivo de status como pai. O arquivo de fornecedores é o arquivo filho para essas duas estruturas.

D.7 TÉCNICAS DE COMPACTAÇÃO

As técnicas de compactação são formas de reduzir o armazenamento necessário para uma determinada coleção de dados. Muitas vezes, o resultado dessa compactação será não só economizar espaço de armazenamento, mas também (e talvez mais significantemente) economizar E/S de disco; pois, se os dados ocuparem menos espaço, menos operações de E/S serão necessárias para acessá-los. Por outro lado, será necessário um processamento extra para descompactar os dados após terem sido buscados. Pesando tudo, no entanto, a economia de E/S provavelmente supera a desvantagem desse processamento adicional.

As técnicas de compactação são elaboradas para explorar o fato de que os valores de dados quase nunca são completamente aleatórios, mas, ao contrário, apresentam uma certa previsibilidade. Como um exemplo simples, se o nome de uma determinada pessoa em um arquivo de nomes e endereços começa

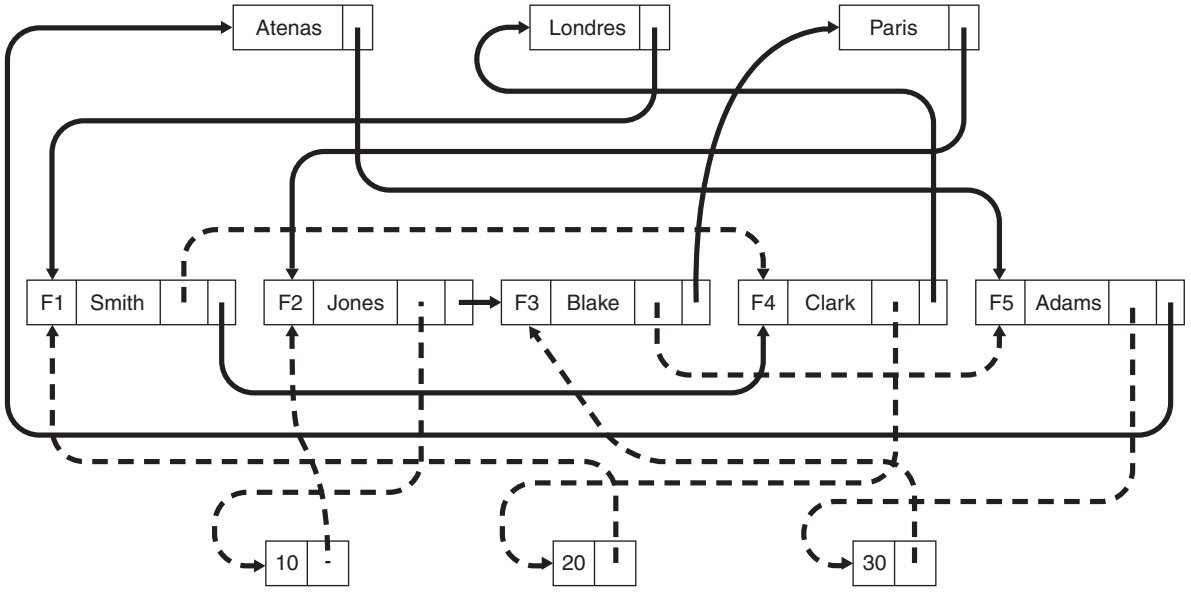

FIGURA D.17 *Outro exemplo de uma estrutura pai/filho*

com a letra R, então, será muito provável que o nome da próxima pessoa também começará com a letra R – considerando, é claro, que o arquivo esteja em ordem alfabética por nome.

Uma técnica de compactação comum, portanto, é substituir cada valor de dados individual por alguma representação da diferença entre ele e o valor que o precede: **compactação diferencial**. Note, entretanto, que essa técnica exige que os dados em questão sejam acessados seqüencialmente, pois, para descompactar qualquer valor armazenado é necessário conhecer o valor armazenado imediatamente anterior. Portanto, a compactação diferencial tem sua principal aplicação em situações em que os dados precisariam mesmo ser acessados seqüencialmente, como, por exemplo, no caso das entradas em um índice de nível único. Note, porém, que, no caso de um índice especificamente, os ponteiros podem ser compactados assim como os dados – pois, se a ordenação de dados lógica imposta pelo índice for igual ou semelhante à ordenação física do arquivo principal, então, os valores de ponteiro sucessivos no índice serão muito semelhantes uns aos outros e a compactação de ponteiro provavelmente será benéfica. Na verdade, os índices quase sempre estão em situação de lucrar com o uso da compactação, pelo menos para os dados, se não para os ponteiros.

Para ilustrar a compactação diferencial, vamos deixar por um momento os fornecedores e peças e consideremos uma página de entradas de um índice de "nomes de funcionários". Suponha que as quatro primeiras entradas nessa página são para os seguintes funcionários:

```
Roberton
Robertson
Robertstone
Robinson
```

Suponha também que os nomes de funcionários possuem 12 caracteres de extensão, de modo que cada um desses nomes deve ser considerado (em sua forma descompactada) como estando preenchido à direita com um número apropriado de lacunas. Uma maneira de aplicar compactação diferencial nesse conjunto de valores é substituindo os caracteres no início de cada entrada que sejam iguais aos da entrada anterior por uma contagem correspondente: **compactação anterior** (front compression). Esse método produz:

```
0 – Roberton++++
6 – son+++
7 – tone+
3 – inson++++
```

(as lacunas finais, agora, aparecem explicitamente como "+").

Outra técnica de compactação possível para esse conjunto de dados é simplesmente eliminar todas as lacunas finais (novamente, substituindo-as por uma contagem apropriada): um exemplo de **compactação posterior** (rear compression). Uma compactação posterior adicional pode ser conseguida descartando todos os caracteres à direita do caractere necessário para distinguir a entrada em questão dos seus dois vizinhos imediatos, da seguinte forma:

```
0 – 7 – Roberto
6 – 2 – so
7 – 1 – t
3 – 1 – i
```

A primeira das duas contagens em cada entrada aqui é como no exemplo anterior; a segunda é uma contagem do número de caracteres registrados (consideramos que a próxima entrada não possui "Robi" como seus quatro primeiros caracteres quando descompactados). Observe, entretanto, que realmente perdemos alguma informação deste índice. Ou seja, quando descompactado, ele se parece com isto:

```
Roberto?????
Robertso????
Robertst????
Robi????????
```

(onde "?" representa um caractere desconhecido). Essa perda de informação, obviamente, só é permitida se os dados forem registrados integralmente em *algum lugar*: no exemplo, no arquivo de funcionários principal.

Compactação hierárquica

Suponha que um determinado arquivo seja seqüenciado fisicamente – isto é, clusterizado – por valores de algum campo *F*, e suponha também que cada valor diferente de *F* ocorre em vários registros consecutivos desse arquivo. Por exemplo, o arquivo de fornecedores pode ser clusterizado por valores do campo de cidade, caso em que todos os fornecedores de Londres seriam armazenados juntos, todos os fornecedores de Paris seriam armazenados juntos e assim por diante. Nessa situação, o conjunto de todos os registros de fornecedor para uma determinada cidade poderia vantajosamente ser compactado em um único registro **hierárquico**, em que o valor de cidade em questão aparece exatamente uma vez, seguido do número de fornecedor, nome e informações de status para cada fornecedor que esteja localizado nessa cidade. Veja a Figura D.18.

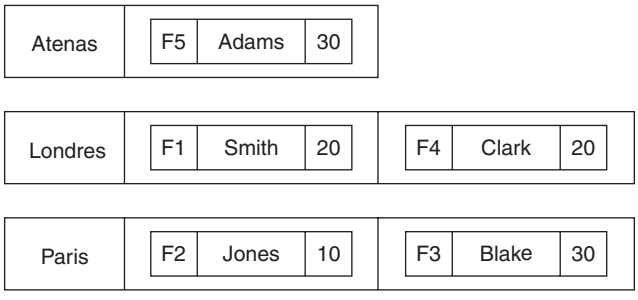

FIGURA D.18 *Exemplo de compactação hierárquica (intra-arquivo)*

Os registros na Figura D.18 consistem em duas partes: uma parte fixa (o campo de cidade) e uma parte variável (o conjunto das entradas de fornecedor). *Nota*: A última parte é variável no sentido de que o número de entradas que ela contém – isto é, o número de fornecedores na cidade em questão – varia de uma ocorrência do registro para outra. Como dissemos no Capítulo 6, esse conjunto variável de entradas den-

tro de um registro normalmente é chamado de *grupo repetido*. Portanto, poderíamos dizer que os registros hierárquicos da Figura D.18 consistem em um único campo de cidade e um grupo repetido de informações de fornecedor, e as informações de fornecedor, por sua vez, consistem em um campo de número de fornecedor, um campo de nome de fornecedor e um campo de status de fornecedor (um grupo de campos para cada fornecedor na cidade relevante).

A compactação hierárquica do tipo descrito anteriormente é, em geral, particularmente apropriada em um índice, onde normalmente acontece de várias entradas sucessivas terem o mesmo valor de dados (embora, é claro, diferentes valores de ponteiro).

Em vista disso, concluímos que a compactação hierárquica do tipo ilustrado é viável apenas se o cluster intra-arquivo estiver em vigor. Porém, como você já pode ter percebido, um tipo semelhante de compactação pode ser aplicado também com cluster *inter*arquivo. Suponha que fornecedores e carregamentos estejam clusterizado como sugerido no final da Seção D.2 – ou seja, carregamentos para o fornecedor F1 seguem imediatamente o registro de fornecedor para F1, carregamentos para o fornecedor F2 seguem imediatamente o registro de fornecedor para F2 e assim por diante. Mais especificamente, suponha que o fornecedor F1 e os carregamentos para o fornecedor F1 estejam armazenados na página *p1*, o fornecedor F2 e os carregamentos para o fornecedor F2 estejam armazenados na página *p2* e assim por diante. Então, uma técnica de compactação interarquivo pode ser aplicada como mostra a Figura D.19.

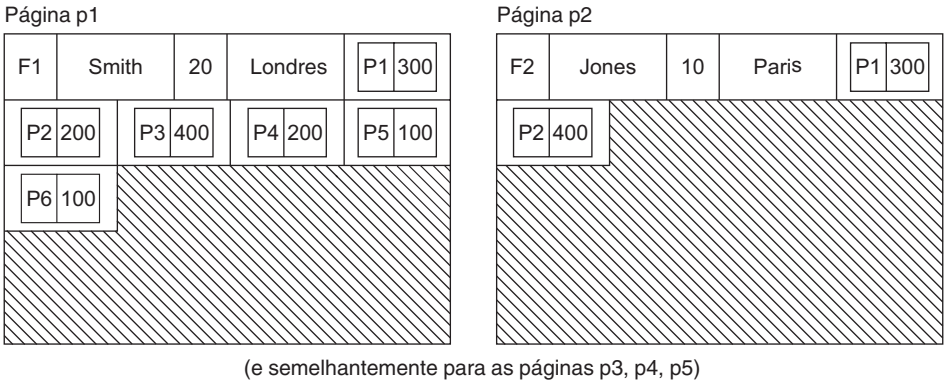

FIGURA D.19 *Exemplo de compactação hierárquica (interarquivo)*

Nota: Embora tenhamos descrito esse exemplo como "interarquivo", na verdade, ele equivale a combinar os arquivos de fornecedor e carregamento em um único arquivo e, depois, aplicar compactação *intra*-arquivo nesse arquivo único. Logo, esse caso não é realmente diferente, em tipo, do caso já ilustrado na Figura D.18.

Concluímos esta subseção observando que a estrutura de cadeia de ponteiros da Figura D.16 pode ser considerada como um tipo de compactação interarquivo que não requer qualquer cluster interarquivo correspondente (ou, mais precisamente, os ponteiros fornecem o efeito lógico desse tipo de cluster – para que a compactação seja possível – mas não necessariamente fornecem a vantagem do desempenho físico correspondente ao mesmo tempo – de modo que a compactação, embora possível, pode não ser uma boa idéia).

Codificação de Huffman

A "codificação de Huffman" [D.39] é uma técnica de codificação de caractere que, embora pouco usada nos sistemas atuais, pode, em princípio, resultar em uma compactação de dados significante se diferentes caracteres ocorrerem nos dados com diferentes freqüências (que é a situação normal, evidentemente). A idéia básica é a seguinte: Codificações de string de bits são atribuídas para representarem caracteres de tal modo que diferentes caracteres sejam representados por strings de bits de extensões diferentes e os caracteres que ocorrem com mais freqüência são representados pelas strings mais curtas. Além disso, nenhum

caractere possui uma codificação (por exemplo, de *n* bits) tal que esses *n* bits sejam idênticos aos *n* primeiros bits de alguma outra codificação de caractere.

Como um exemplo simples, imagine que os dados a serem representados envolvem apenas os caracteres A, B, C, D e E, e que a freqüência relativa de ocorrência desses cinco caracteres seja como indicado na tabela a seguir:

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

O caractere E possui a freqüência mais alta e, portanto, lhe é atribuído o código mais curto, por exemplo, um bit 1. Todos os outros códigos precisam, então, iniciar com um 0 bit e precisam ter, pelo menos, 2 bits de extensão (um bit 0 isolado não seria válido, já que ele seria indistinguível da parte inicial dos outros códigos). Ao caractere A é atribuído o próximo código mais curto, digamos, 01; todos os outros códigos, portanto, precisam iniciar com 00. Da mesma forma, aos caracteres D, C e B é atribuído os códigos 001, 0001 e 0000, respectivamente. *Exercício*: Que palavras em inglês as seguintes strings representam?

```
00110001010011
010001000110011
```

Dadas as codificações mostradas, a extensão média esperada de um caractere codificado, em bits, é

```
0.35 * 1 + 0.30 * 2 + 0.20 * 3 + 0.10 * 4 + 0.05 * 4 = 2.15 bits,
```

enquanto, se um mesmo número de bits fosse atribuído a cada caractere, como em um esquema de codificação de caracteres convencional, precisaríamos de 3 bits por caractere (para considerar as cinco possibilidades).

D.8 RESUMO

Neste apêndice, tivemos uma extensa – mas de modo algum exaustiva! – análise de algumas das estruturas de armazenamento mais comumente encontradas na prática atual. Também descrevemos superficialmente como os softwares de acesso a dados geralmente funcionam e rascunhamos as maneiras em que a responsabilidade é dividida entre o **SGBD**, o **gerenciador de arquivos** e o **gerenciador de discos**. Nosso objetivo, o tempo todo, foi explicar conceitos gerais, não descrever em profundidade como os vários componentes de sistema e estruturas de armazenamento realmente funcionam. Com certeza, procuramos não entrar em muitos detalhes, embora um certo nível de detalhamento naturalmente seja inevitável.

Como resumo, aqui está uma breve revisão de alguns dos principais assuntos abordados. Descrevemos o **cluster**, cuja idéia básica é que os registros que são usados juntos devem ser armazenados fisicamente próximos. Também explicamos como os registros são identificados internamente pelos **IDs de registro (RIDs)**. Em seguida, Consideramos algumas das estruturas de armazenamento mais importantes encontradas na prática:

- **Índices** (diversas variações deles, incluindo especialmente **árvore-b**) e seu uso para acesso **seqüencial** e **direto**
- **Hashing** (incluindo especialmente o hashing **extensível**) e seu uso para acesso **direto**
- **Cadeias de ponteiros** (também conhecidas como **estruturas pai/filho**) e diversas variações delas

Também examinamos uma variedade de técnicas de **compactação**.

Concluímos enfatizando o fato de que a maioria dos usuários é (ou deveria ser) despreocupada com a maioria desses assuntos na maior parte do tempo. O único "usuário" que precisa entender esses conceitos em detalhes é o DBA, que é responsável pelo projeto físico do banco de dados e por realizar monitoramento e ajustes. Para os outros usuários, essas considerações devem, preferivelmente, estar ocultas da visão, embora talvez se possa dizer que esses usuários realizarão um trabalho melhor se tiverem noção de como o sistema funciona internamente. Para implementadores de SGBD, por outro lado, um conhecimento prático desses assuntos (na verdade, um conhecimento que vai muito além deste estudo introdutório) certamente seja desejável, se não essencial.

EXERCÍCIOS

Os Exercícios D.1 a D.8 podem se mostrar adequados como uma base para debates em grupo; eles têm o objetivo de levar a um entendimento mais aprofundado dos vários aspectos físicos do projeto de banco de dados. Os Exercícios D.9 e D.10, por sua vez, possuem uma característica matemática.

D.1 Investigue quaisquer sistemas de banco de dados (quanto maior, melhor) que possam estar disponíveis a você. Para cada um desses sistemas, identifique os componentes que realizam as funções descritas no corpo deste apêndice, em relação, respectivamente, ao gerenciador de discos, gerenciador de arquivos e o próprio SGBD. Que tipos de discos ou outros meios de armazenamento o sistema suporta? Que tamanhos de página? Quais são as capacidades de disco, tanto teóricas (em bytes) quanto reais (em páginas)? Quais são as taxas de dados? E os tempos de acesso? Como esses tempos de acesso se comparam com a velocidade da memória principal? Existem limites para o tamanho de arquivo ou tamanho de banco de dados? Se sim, quais são eles? Quais das estruturas de armazenamento descritas neste apêndice o sistema suporta? Ele suporta outras estruturas? Caso suporte, quais são elas?

D.2 O banco de dados de pessoal de uma empresa deve conter informações sobre as divisões, departamentos e funcionários dessa empresa. Cada funcionário trabalha em um departamento; cada departamento é parte de uma divisão. Crie alguns dados de exemplo e esboce algumas estruturas de armazenamento correspondentes possíveis. Onde possível, cite as vantagens relativas de cada uma dessas estruturas – ou seja, considere como as operações comuns de busca e atualização seriam manipuladas em cada caso. *Sugestão*: As restrições "cada funcionário trabalha em um departamento" e "cada departamento é parte de uma divisão" são estruturalmente semelhantes à restrição "cada fornecedor está situado em uma cidade" (todas elas são exemplos de relações muitos para um). Uma diferença é que provavelmente gostaríamos de registrar mais informações no banco de dados para departamentos e divisões do que para cidades.

D.3 Repita o Exercício D.2 para um banco de dados que deve conter informações sobre clientes e itens. Cada cliente pode pedir qualquer número de itens e cada item pode ser pedido por qualquer número de clientes. *Sugestão*: Aqui, há uma relação muitos para muitos entre clientes e itens. Uma maneira de representar essa relação é por meio de um *índice duplo*. Um índice duplo é um índice usado para indexar dois arquivos de dados simultaneamente; uma determinada entrada nesse índice corresponde a um par de registros de dados relacionados, um de cada um dos dois arquivos, e contém dois valores de dados e dois ponteiros. Você consegue imaginar outras maneiras de representar relações muitos para muitos(-para muitos-...)?

D.4 Repita o Exercício D.2 para um banco de dados que deve conter informações sobre peças e componentes, onde um componente é, ele próprio, uma peça e pode ter componentes de nível inferior. *Dica*: Em que esse problema difere do problema do Exercício D.3?

D.5 Um arquivo de registros de dados sem qualquer estrutura de acesso adicional é, algumas vezes, chamado de **heap**. Novos registros são inseridos em um heap em qualquer lugar onde haja espaço. Para arquivos pequenos – certamente, para qualquer arquivo que não exija mais do que nove ou dez páginas de armazenamento – um heap provavelmente é a estrutura mais eficiente de todas. Porém, a maioria dos arquivos é maior do que isso e, na prática, todos exceto os menores arquivos devem ter alguma estrutura de acesso adicional, como (pelo menos) um índice na chave primária. Cite as vantagens e desvantagens de uma estrutura indexada em relação a uma estrutura de heap.

D.6 No corpo deste apêndice, fizemos várias referências a cluster físico. Por exemplo, pode ser vantajoso armazenar os registros de fornecedor de modo que sua seqüência física seja igual ou semelhante à sua seqüência lógica como definida pelos valores do campo de número de fornecedor (o *campo de cluster*). Como o SGBD pode fornecer esse cluster físico?

D.7 Na Seção D.5, sugerimos que um método de manipular colisões de hash seria tratar a saída da função de hash como o ponto de partida para uma leitura seqüencial (a técnica de *pesquisa linear*). Você pode ver alguma dificuldade com esse esquema?

D.8 Quais são as vantagens e desvantagens relativas da organização pai/filho múltipla? (Veja o final da Seção D.6. Ela pode ajudar a rever as desvantagens relativas da organização de índice múltipla. Quais são as semelhanças? E as diferenças?)

D.9 Vamos definir "indexação completa" para indicar que existe um índice para cada combinação de campo diferente (e diferentemente ordenado) no arquivo indexado. Por exemplo, a indexação completa para um arquivo com dois campos *A* e *B* exigiria dois índices: um na combinação *AB* (nesta ordem) e um na combinação *BA* (nesta ordem). Quantos índices são necessários para fornecer indexação completa para um arquivo definido em (a) 3 campos, (b) 4 campos e (c) *N* campos?

D.10 Considere uma b-tree simplificada (conjunto de índice mais conjunto de seqüência) em que o conjunto de seqüência contém um ponteiro para cada um dos *N* registros de dados e cada nível acima do conjunto de seqüência (ou seja, cada nível do conjunto de índice) contém um ponteiro para cada página no nível abaixo. Naturalmente, no nível superior (raiz), existe uma única página. Suponha também que cada página do conjunto de índice contém *n* entradas de índice. Derive expressões para o número de *níveis* e o número de *páginas* em toda a b-tree.

D.11 Os primeiros 10 valores do campo indexado em um certo arquivo indexado são os seguintes:

```
Abrahams,GK
Ackermann,LZ
Ackroyd,S
Adams,T
Adams,TR
Adamson,CR
Allen,S
Ayres,ST
Bailey,TE
Baileyman,D
```

Cada um deles é preenchido com lacunas à direita até uma extensão total de 15 caracteres. Mostre os valores realmente registrados no *índice* se forem aplicadas as técnicas de compactação anterior e posterior descritas na Seção D.7. Qual é a economia de espaço em porcentagem? Mostre as etapas envolvidas na busca (ou tentativa de busca) dos registros para "Ackroyd,S" e "Adams,V". Mostre também as etapas envolvidas na inserção de um novo registro para "Allingham,M".

REFERÊNCIAS E BIBLIOGRAFIA

As referências a seguir estão organizadas em grupos, da seguinte maneira: As referências [D.1-D.10] são livros didáticos exclusivamente dedicados ao tópico deste apêndice ou, pelo menos, apresentam um tratamento detalhado dele. As referências [D.11–D.15] descrevem alguns métodos formais para o assunto. As referências [D.16–D.23] estão voltadas especificamente à indexação, em especial, às árvore-b; as referências [D.24–D.38] representam uma seleção da extensa literatura sobre hashing; as referências [D.39–D.40] discutem as técnicas de compactação; e, finalmente, as referências [D.41–D.59] tratam de algumas estruturas de armazenamento gerais e questões relacionadas (em particular, as referências [D.48–D.59] abordam certos meios de armazenamento mais recentes e novos tipos de aplicações, bem como as estruturas de armazenamento para esses meios e aplicações).

D.1 Donald E. Knuth: The Art of Computer Programming. Volume III: Sorting and Searching (2ª ed.). Reading, Mass.: Addison-Wesley (1998).

O Volume III da série clássica de volumes de Knuth contém uma análise completa dos algoritmos de pesquisa. Para pesquisa de *banco de dados*, onde os dados residem no armazenamento secundário, as seções que se aplicam mais diretamente são 6.2.4 (Multi-way Trees), 6.4 (Hashing) e 6.5 (Retrieval on Secondary Keys).

D.2 James Martin: *Computer Data-Base Organization* (2ª ed.). Englewood Cliffs, N.J.: Prentice-Hall (1977).

Este livro é dividido em duas partes principais, "Logical Organization" e "Physical Organization". A última parte consiste em uma extensa descrição (mais de 300 páginas) das estruturas de armazenamento e as técnicas de acesso correspondentes.

D.3 (O mesmo que a referência [14.45].) Toby J. Teorey e James P. Fry: *Design of Database Structures*. Englewood Cliffs, N.J.: Prentice-Hall (1982).

Um tutorial e livro de bolso sobre projeto físico e lógico de bancos de dados. Mais de 200 páginas são dedicadas ao projeto físico.

D.4 Gio Wiederhold: *Database Design* (2ª ed.). Nova York, N.Y.: McGraw-Hill (1983).

Este livro de 15 capítulos inclui um bom estudo dos dispositivos de armazenamento secundários e seus parâmetros de desempenho (um capítulo, quase 50 páginas) e uma extensa análise das estruturas de armazenamento secundárias (quatro capítulos, mais de 250 páginas).

D.5 T. H. Merrett: *Relational Information Systems*. Reston, Va.: Reston Publishing Company, Inc. (1984).

Inclui uma extensa introdução (e análise) à variedade de estruturas de armazenamento (cerca de 100 páginas), abordando não só as estruturas descritas neste apêndice, mas também várias outras.

D.6 Jeffrey D. Ullman: *Principles of Database and Knowledge-Base Systems: Volume I*. Rockville, Md.: Computer Science Press (1988).

Inclui uma abordagem das estruturas de armazenamento que é muito mais teórica do que a deste apêndice.

D.7 (O mesmo que a referência [16.21].) Abraham Silberschatz, Henry F. Korth e S. Sudarshan: *Database System Concepts* (4ª ed.). Nova York, N.Y.: McGraw-Hill (2002).

D.8 Peter D. Smith e G. Michael Barnes: *Files and Databases: An Introduction*. Reading, Mass.: Addison-Wesley (1987).

D.9 (O mesmo que a referência [14.18].) Ramez Elmasri and Shamkant B. Navathe: *Fundamentals of Database Systems* (3ª ed.). Redwood City, Calif.: Benjamin/Cummings (2000).

As referências [D.7], [D.8] e [D.9] são livros didáticos sobre sistemas de banco de dados. Cada um inclui material sobre estruturas de armazenamento que vai além da abordagem deste apêndice em certos aspectos (especialmente no caso da referência [D.8]).

D.10 Sakti P. Ghosh: *Data Base Organization for Data Management* (2ª ed.). Orlando, Fla.: Academic Press (1986).

A ênfase principal deste livro está nas estruturas de armazenamento e nos métodos de acesso associados (dos dez capítulos do livro, pelo menos seis são dedicados a esses tópicos). A abordagem é bastante abstrata.

D.11 David K. Hsiao e Frank Harary: "A Formal System for Information Retrieval from Files", *CACM 13,* Número 2 (fevereiro de 1970).

Este documento representa o que provavelmente foi a primeira tentativa de unificar as idéias de diferentes estruturas de armazenamento – principalmente índices e cadeias de ponteiros – em um único modelo geral, fornecendo, assim, uma base para uma teoria formal dessas estruturas. Um algoritmo de busca genérico é apresentado para buscar registros da estrutura geral que satisfaz uma combinação booleana arbitrária de condições "*campo = valor*".

D.12 Dennis G. Severance: "Identifier Search Mechanisms: A Survey and Generalized Model", *ACM Comp. Surv. 6,* Número 3 (setembro de 1974).

Este documento é dividido em duas partes. A primeira parte fornece um tutorial sobre certas estruturas de armazenamento (basicamente, hashing e indexação). A segunda parte possui pontos em comum com a referência [D.11]; assim como aquele ensaio, este documento define uma estrutura unificada, aqui chamada estrutura *trie-tree*, que combina e generaliza idéias das estruturas discutidas na primeira parte. (O termo *trie* deriva de um estudo de Fredkin [D.42].) A estrutura resultante fornece um modelo geral que pode representar uma ampla variedade de estruturas diferentes em termos de um pequeno número de parâmetros; ele pode, então, ser usado (e realmente tem sido usado) para ajudar a escolher uma estrutura específica durante o processo de projeto físico do banco de dados.

Uma diferença entre este documento e a referência [D.11] é que a estrutura trie-tree manipula hashes mas não cadeias de ponteiros, enquanto a proposta da referência [D.11] manipula cadeias de ponteiros mas não hashes.

D.13 M. E. Senko, E. B. Altman, M. M. Astrahan e P. L. Fehder: "Data Structures and Accessing in Data-Base Systems", *IBM Sys. J. 12,* Número 1 (1973).

Este documento se divide em três partes:

1. Evolução de sistemas de informações
2. Organização de informações
3. Representações de dados e o modelo de acesso independente aos dados

A primeira parte consiste em uma análise histórica do desenvolvimento dos sistemas de banco de dados antes de 1973. A segunda parte descreve "o modelo de conjunto de entidades", que fornece uma base para descrever uma determinada empresa em termos de entidades e conjuntos de entidades (ele corresponde ao nível conceitual da arquitetura ANSI/SPARC). A terceira parte é a mais original e significante do documento; ela forma uma introdução ao *Data Independent Accessing Model* (DIAM), que é uma tentativa de descrever um banco de dados em termos de quatro níveis de abstração sucessivos: o conjunto de entidades (o nível mais alto), string, codificação e dispositivo físico. Esses quatro níveis podem ser imaginados como uma definição mais detalhada, mas ainda abstrata, das partes conceituais e internas da arquitetura ANSI/SPARC. Eles podem ser brevemente descritos desta forma:

- *Nível de conjunto de entidades*: Análogo ao nível conceitual ANSI/SPARC.
- *Nível de string*: Os caminhos de acesso aos dados são definidos como conjuntos ordenados ou "strings" de objetos de dados. Três tipos de strings são descritos: strings atômicas (por exemplo, uma string conectando ocorrências de campo para formar uma ocorrência de registro de peça), strings de entidade (por exemplo, uma string conectando ocorrências de registro de peça para peças vermelhas) e strings de link (por exemplo, uma string conectando uma ocorrência de registro de fornecedor a ocorrências de registro de peça para peças vendidas por esse fornecedor).
- *Nível de codificação*: Os objetos de dados e strings são mapeados para espaços de endereço linear, usando uma primitiva de representação simples chamada *unidade de codificação básica*.
- *Nível de dispositivo físico*: Os espaços de endereço linear são alocados para subdivisões físicas formatadas de meios de registro reais.

O objetivo do DIAM, assim como as referências [D.11–D.12], é, em parte, fornecer a base para uma teoria sistemática de estruturas de armazenamento e métodos de acesso. Uma crítica – que, por acaso, também se aplica aos formalismos das referências [D.11] e [D.12] – é que, algumas vezes, o melhor método de lidar com alguma requisição de acesso específica é simplesmente classificar os dados, e o armazenamento é, naturalmente, dinâmico, enquanto as estruturas descritas pelo DIAM (e os modelos das referências [D.11] e [D.12]) são, por definição, sempre estáticos.

D.14 S. B. Yao: "An Attribute Based Model for Database Access Cost Analysis", *ACM TODS 2,* Número 1 (março de 1977).

A finalidade deste documento é semelhante à das referências [D.11] e [D.12]; em alguns aspectos, na verdade, ele pode ser considerado uma continuação dos ensaios anteriores, na medida em que ele representa um modelo generalizado de estruturas de armazenamento que pode ser visto como uma combinação e extensão das propostas daqueles documentos. São feitas referências a diversos outros documentos que relatam experimentos com um analisador de projeto de arquivo físico implementado com base nas idéias deste documento.

D.15 D. S. Batory: "Modeling the Storage Architectures of Commercial Database Systems", *ACM TODS 10,* Número 4 (dezembro de 1985).

Apresenta um conjunto de operações primitivas, chamadas *transformações elementares*, pelo qual o mapeamento do esquema conceitual para o esquema interno correspondente (isto é, o mapeamento conceitual/interno – veja o Capítulo 2) pode se tornar explícito e, portanto, ser adequadamente estudado. As transformações elementares incluem *ampliação* (estender um registro pela inclusão de dados de prefixo além dos dados de usuário), *codificação* (converter dados em uma forma interna através, por exemplo,

de compactação), *segmentação* (dividir um registro em várias partes para fins de armazenamento) e várias outras. O documento propõe que qualquer mapeamento conceitual/interno pode ser representado por uma seqüência apropriada dessas transformações elementares e, conseqüentemente, que as transformações poderiam formar a base de um método para automatizar o desenvolvimento de softwares de gerenciamento de dados. Através de ilustração, o documento aplica as idéias à análise de três sistemas comerciais: INQUIRE, ADABAS e System 2000. *Nota:* Compare e confronte as propostas GMAP (muito posteriores) da referência [2.5]. Consulte também o Apêndice A para ver alguns contra-exemplos possíveis.

D.16 R. Bayer e C. McCreight: "Organization and Maintenance of Large Ordered Indexes", *Acta Informatica* 1, Número 3 (1972).

D.17 Douglas Comer: "The Ubiquitous B-Tree", *ACM Comp. Surv. 11,* Número 2 (junho de 1979).

Um bom tutorial sobre b-trees.

D.18 R. E. Wagner: "Indexing Design Considerations", *IBM Sys. J. 12,* Número 4 (1973).

Descreve conceitos básicos de indexação, com detalhes das técnicas – incluindo técnicas de compactação – usadas no IBM's Virtual Storage Access Method (VSAM).

D.19 H. K. Chang: "Compressed Indexing Method", *IBM Technical Disclosure Bulletin II,* Número 11 (abril de 1969).

D.20 Gopal K. Gupta: "A Self-Assessment Procedure Dealing with Binary Search Trees and B-Trees", *CACM 27,* Número 5 (maio de 1984).

D.21 Vincent Y. Lum: "Multi-attribute Retrieval with Combined Indexes", *CACM 13,* Número 11 (novembro de 1970).

O documento que introduziu a técnica de indexar em combinações de campo.

D.22 James K. Mullin: "Retrieval-Update Speed Tradeoffs Using Combined Indices", *CACM 14,* Número 12 (dezembro de 1971).

Uma continuação à referência [D.21] que fornece estatísticas de desempenho para o esquema de índice combinado para várias relações busca/atualização.

D.23 Ben Shneiderman: "Reduced Combined Indexes for Efficient Multiple Attribute Retrieval", *Information Systems 2*, Número 4 (1976).

Propõe um aprimoramento da técnica de indexação combinada de Lum [D.21] que reduz consideravelmente o espaço de armazenamento e a sobrecargas de tempo de busca. Por exemplo, a combinação de índice *ABCD, BCDA, CDAB, DABC, ACBD, BDAC* – veja a resposta ao Exercício D.9(b) – poderia ser substituída pela combinação *ABCD, BCD, CDA, DAB, AC, BD.* Se cada um de *A, B, C, D* puder assumir 10 valores diferentes, então, na pior hipótese, a combinação original envolveria 60.000 entradas de índice, enquanto a combinação reduzida envolveria apenas 13.200 entradas.

D.24 R. Morris: "Scatter Storage Techniques", *CACM 11,* Número 1 (janeiro de 1968).

Este documento focaliza principalmente o hashing já que ele se aplica à tabela de símbolos de um assembler ou compilador. Sua finalidade principal é descrever um esquema de hashing **indireto** baseado em *tabelas dispersas.* Uma tabela dispersa é uma tabela de endereços de registro, de certa forma semelhante ao diretório usado no hashing extensível [D.28]. Assim como no hashing extensível, a função de hash realiza hash na tabela dispersa, não diretamente nos próprios registros; os registros propriamente ditos podem ser armazenados em qualquer lugar que pareça conveniente. A tabela dispersa, portanto, pode ser imaginada como um *índice* de nível único para os dados básicos, mas um índice que pode ser acessado diretamente através de um hash em vez de precisar ser seqüencialmente pesquisado. Note que um determinado arquivo de dados poderia, conceitualmente, ter várias tabelas dispersas diferentes, assim, efetivamente fornecendo acesso de hash aos dados em vários campos de hash distintos (à custa de um E/S extra para qualquer acesso de hash).

Apesar de estar orientado para a linguagem de programação, o documento oferece uma boa introdução às técnicas de hashing em geral, e a maioria do material também é aplicável ao hashing de banco de dados.

D.25 W. D. Maurer e T. G. Lewis: "Hash Table Methods", *ACM Comp. Surv. 7,* Número 1 (março de 1975).

Um bom tutorial, embora um pouco desatualizado hoje (ele não discute qualquer um dos métodos mais recentes, como hashing extensível). Os assuntos abordados incluem técnicas básicas de hashing (não só

divisão/resto, mas também técnicas de análise aleatória, raiz, codificação algébrica, desdobramento e dígitos); tratamento de colisão e estouro de bucket; alguma análise teórica das várias técnicas; e alternativas ao hashing (técnicas a serem usadas quando o hashing não pode ou não deve ser usado). *Nota*: Um **bucket** na terminologia de hashing é a unidade de armazenamento – em geral, uma página – cujo endereço é calculado pela função de hash. Um bucket normalmente contém vários registros.

D.26 V. Y. Lum, P.S.T. Yuen e M. Dodd: "Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files", *CACM 14,* Número 4 (abril de 1971).

Uma investigação sobre o desempenho de vários algoritmos de hashing "básicos" (ou seja, não extensíveis). A conclusão é que o método divisão/resto parece ser o melhor de todos.

D.27 M. V. Ramakrishna: "Hashing in Practice: Analysis of Hashing and Universal Hashing", Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, Ill. (junho de 1988).

Como este documento destaca (seguindo Knuth [D.1]), qualquer sistema que implementa hashing precisa resolver dois problemas que são quase independentes um do outro: (a) ele precisa escolher, dentre a grande variedade de funções de hash disponíveis, uma que seja eficaz e (b) ele também precisa fornecer uma técnica eficaz para lidar com colisões. O autor afirma que, embora muito estudo tem sido dedicado ao segundo desses problemas, muito pouco tem sido realizado sobre o primeiro, e poucas tentativas foram feitas para comparar o desempenho do hashing, na prática, com o desempenho que, na teoria, pode ser obtido (a referência [D.26] é uma exceção). Como, então, um implementador de sistema deve escolher uma função de hash apropriada? Este documento afirma que é possível escolher uma função de hash que, na prática, realmente produz um desempenho semelhante ao previsto pela teoria e apresenta um grupo de resultados teóricos em apoio a essa afirmação.

D.28 Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger e H. Raymond Strong: "Extendible Hashing – A Fast Access Method for Dynamic Files", *ACM TODS 4,* Número 3 (setembro de 1979).

D.29 G. D. Knott: "Expandable Open Addressing Hash Table Storage and Retrieval", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif. (novembro de 1971).

D.30 P.-Å. Larson: "Dynamic Hashing", *BIT 18* (1978).

D.31 Witold Litwin: "Virtual Hashing: A Dynamically Changing Hashing", Proc. 4th Int. Conf. on Very Large Data Bases, Berlim, Alemanha (setembro de 1978).

D.32 Witold Litwin: "Linear Hashing: A New Tool for File and Table Addressing", Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canadá (outubro 1980).

D.33 Per-Åke Larson: "Linear Hashing with Overflow-Handling by Linear Probing", *ACM TODS 10,* Número 1 (março de 1985).

D.34 Per-Åke Larson: "Linear Hashing with Separators – A Dynamic Hashing Scheme Achieving One-Access Retrieval", *ACM TODS 13,* Número 3 (setembro de 1988).

As referências [D.28–D.34] apresentam esquemas de hashing extensível de um tipo ou de outro. As propostas de [D.29] para hashing "expansível" são anteriores a (e, portanto, bastante independentes de) todas as outras. Porém, o hashing expansível é muito semelhante ao hashing extensível, como definido na referência [D.28]. O hashing "virtual" [D.31] é um pouco diferente; veja o documento para obter detalhes. O hashing "linear", introduzido em [D.32] e aprimorado em [D.33] e [D.34], é uma evolução do hashing virtual.

D.35 Witold Litwin: "Trie Hashing", Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data, Ann Arbor, Mich. (abril de 1981).

Apresenta um esquema de hashing extensível com várias propriedades desejáveis:

- Ele é *preservador de ordem* (ou seja, a seqüência "física" dos registros corresponde à seqüência lógica desses registros como definidos pelos valores do campo de hash).
- Ele evita os problemas de complexidade, entre outros, normalmente encontrados nos hashes preservadores de ordem.
- Um registro arbitrário pode ser acessado (ou revelado não existir) em um único acesso ao disco, mesmo se o arquivo contiver muitos milhões de registros.

- O arquivo pode ser arbitrariamente volátil (por outro lado, muitos esquemas de hash, pelo menos do tipo não extensível, tendem a funcionar muito mal diante de altos volumes de inserção).

A função de hash propriamente dita (que muda com o tempo, como em todos os algoritmos de hashing extensível) é representada por uma estrutura trie [D.42], que é mantida na memória principal sempre que o arquivo está em uso e cresce elegantemente à medida que cresce o arquivo de dados. O arquivo de dados em si, como já mencionado, é mantido em seqüência "física" nos valores do campo de hash; e a seqüência lógica das entradas de folha na estrutura trie corresponde, precisamente, à seqüência "física" dos registros de dados. O estouro no arquivo de dados é tratado através de uma técnica de divisão de página, basicamente como a técnica de divisão de página usada em uma árvore-b.

O hashing trie parece muito interessante. Assim como outros esquemas de hash, ele oferece um desempenho melhor do que indexação para acesso direto (um E/S contra dois ou três para a árvore-b); e é preferível para a maioria dos outros esquemas de hash, na medida em que é preservador de ordem, o que significa que o acesso seqüencial também será rápido. Nenhuma árvore-b ou outra estrutura adicional é necessária para fornecer esse acesso seqüencial rápido. Porém, deve-se supor que o trie será incorporado à memória principal (uma suposição provavelmente bastante realista). Se essa suposição for inválida – isto é, se o arquivo de dados for grande demais – ou se a propriedade de preservação de ordem não for necessária, então, o hashing linear [D.32] ou alguma outra técnica poderá fornecer uma alternativa preferível.

D.36 David B. Lomet: "Bounded Index Exponential Hashing", *ACM TODS 8,* Número 1 (março de 1983).

Outro esquema de hashing extensível. O documento afirma que:

- O esquema fornece acesso direto a qualquer registro próximo de um E/S em média (e nunca mais do que dois).
- Ele oferece um desempenho que é independente do tamanho de arquivo. (Por outro lado, a maioria dos esquemas de hashing extensível sofre de degradação de desempenho temporária no momento em que ocorre uma divisão de página de diretório, pois, geralmente, todas essas páginas precisam ser divididas quase ao mesmo tempo.)
- Ele faz uso eficiente do espaço disponível em disco (ou seja, a utilização do espaço pode ser muito boa).
- Sua implementação é simples.

D.37 Anil K. Garg e C. C. Gotlieb: "Order-Preserving Key Transformations", *ACM TODS 11,* Número 2 (junho de 1986).

Como explicado na anotação da referência [D.35], uma função de hash preservadora de ordem (ou transformação de chave") é uma em que a seqüência física dos registros corresponde à seqüência lógica desses registros como definido pelos valores do campo de hash. Os hashes preservadores de ordem são desejáveis por questões óbvias. Uma função simples claramente preservadora de ordem é a seguinte:

```
endereço de hash = quociente após divisão do valor do campo de hash
                   por alguma constante (por exemplo, 10.000)
```

Porém, um problema óbvio com uma função como essa é que ela funciona muito mal se valores do campo de hash forem distribuídos desigualmente (que, naturalmente, é o caso mais comum). Portanto, alguns pesquisadores propuseram o conceito de funções de hash *dependentes de distribuição* (mas preservadoras de ordem), ou seja, funções que transformam valores de campo de hash desigualmente distribuídos em endereços de hash uniformemente distribuídos enquanto mantêm a propriedade preservadora de ordem. (*Nota*: O hashing trie [D.35] é um exemplo desse método.) Este documento fornece um método para construir essas funções de hash para arquivos de dados do mundo real e demonstra a viabilidade prática dessas funções.

D.38 M. V. Ramakrishna e Per-Åke Larson: "File Organization Using Composite Perfect Hashing", *ACM TODS 14,* Número 2 (junho de 1989).

Uma função de hash é chamada de *perfeita* se não produzir estouro algum. (*Nota*: "Nenhum estouro" não significa "nenhuma colisão". Por exemplo, se considerarmos que a função de hash gera endereços de página, não endereços de registro – veja a observação sobre "buckets" na anotação à referência [D.25] – e se cada página pode conter *n* registros, então, a função de hash será perfeita se nunca mapear mais do que *n* registros para a mesma página.) Uma função de hash perfeita tem a propriedade de que qualquer registro pode ser buscado em um único E/S de disco. Este documento apresenta um método prático para encontrar e usar essas funções perfeitas.

D.39 D. A. Huffman: "A Method for the Construction of Minimum Redundancy Codes", Proc. IRE 40 (setembro de 1952).

D.40 B. A. Marron e P. A. D. de Maine: "Automatic Data Compression", *CACM 10,* Número 11 (novembro de 1967).

Fornece dois algoritmos de compactação/descompactação: NUPAK, que opera em dados numéricos, e ANPAK, que opera em dados alfanuméricos ou "quaisquer" dados (ou seja, qualquer string de bits).

D.41 Dennis G. Severance e Guy M. Lohman: "Differential Files: Their Application to the Maintenance of Large Databases", *ACM TODS 1,* Número 3 (setembro de 1976).

Discute os "arquivos diferenciais" e suas vantagens. A idéia básica é que as atualizações não sejam feitas diretamente no próprio banco de dados, mas sejam registradas em um arquivo fisicamente separado – o arquivo diferencial – e sejam mescladas com o banco de dados real em algum momento subseqüente adequado. As vantagens a seguir são alegadas nesse método:

- Os custos de dumping de banco de dados são reduzidos.
- O dumping incremental é facilitado.
- O dumping e a reorganização podem ser realizados simultaneamente às operações de atualização.
- A recuperação após uma falha de programa de aplicação é rápida.
- A recuperação após uma falha de hardware é rápida.
- O risco de uma perda de dados séria é reduzido.
- Os arquivos memo são admitidos eficientemente (veja a explicação a seguir).
- O desenvolvimento de software é simplificado.
- O software do arquivo principal é simplificado.
- Os custos de armazenamento futuros podem ser reduzidos.

Nota: Um "arquivo memo" é uma espécie de cópia de rascunho de alguma parte do banco de dados, usada para fornecer acesso rápido aos dados que, provavelmente, estão atualizados e corretos, mas não oferecem garantia de estarem assim. Veja o assunto dos *snapshots* no Capítulo 10.
Um problema não discutido é o de suportar acesso seqüencial eficiente aos dados – por exemplo, através de um índice – quando alguns dos registros estão no banco de dados real e alguns estão no arquivo diferencial.

D.42 E. Fredkin: "TRIE Memory", *CACM 3,* Número 9 (setembro de 1960).

Um **trie** é um arquivo de dados estruturado em árvore (em vez de um caminho de acesso estruturado em árvore para esse arquivo; ou seja, os dados são representados *pela* árvore, e não apontados *da* árvore – a menos que o "arquivo de dados" seja realmente um índice para algum outro arquivo, já que ele efetivamente está em hashing trie [D.35]). Cada nó em um trie consiste logicamente em *n* entradas, onde *n* é o número de símbolos distintos disponíveis para representar valores de dados. Por exemplo, se cada item de dados for um inteiro decimal, então, cada nó terá exatamente 10 entradas, correspondentes aos dígitos decimais 0, 1, 2, ..., 9. Considere o item de dados "4285". O nó (único) no alto da árvore incluirá um ponteiro na entrada "4". Esse ponteiro apontará para um nó correspondente a todos os itens de dados existentes tendo "4" como seu primeiro dígito. Esse nó (o "nó 4"), por sua vez, incluirá um ponteiro em sua entrada "2" para um nó correspondente a todos os itens de dados tendo "42" como seus dois primeiros dígitos (o "nó 42"). O nó "42" terá um ponteiro em sua entrada "8" para o nó "428" e assim por diante. E se, por exemplo, não houver quaisquer itens de dados iniciando com "429", então, a entrada "9" no nó "42" será vazia (não haverá ponteiro algum); em outras palavras, a árvore é reduzida para conter apenas nós que não sejam vazios. (Portanto, um trie geralmente não é uma árvore equilibrada.)
Nota: O termo *trie* deriva de "retrieval" (busca), mas normalmente é pronunciado como *try*. Os tries também são conhecidos como *árvores de pesquisa raiz* ou *árvores de pesquisa digital*.

D.43 Eugene Wong e T. C. Chiang: "Canonical Structure in Attribute Based File Organization", *CACM 14,* Número 9 (setembro de 1971).

Propõe uma nova estrutura de armazenamento baseada na álgebra booleana. Considera-se que todas as requisições de acesso sejam expressas como uma combinação booleana de condições elementares "*cam-*

po = valor" e que essas condições elementares sejam bem conhecidas. Portanto, um arquivo pode ser particionado em subconjuntos separados para fins de armazenamento. Os subconjuntos são os "átomos" da álgebra booleana, consistindo no conjunto de todos os conjuntos recuperáveis através das requisições de acesso booleanas originais. As vantagens desse arranjo incluem as seguintes:

- Nunca é necessário definir inserção de átomos.
- Uma requisição booleana arbitrária pode ser facilmente convertida em uma requisição para a união de um ou mais átomos.
- Essa união nunca exige a eliminação de duplicatas.

D.44 Michael Stonebraker: "Operating System Support for Database Management", *CACM 24,* Número 7 (julho de 1981).

Discute as razões porque vários recursos de sistema operacional – sobretudo os serviços de gerenciamento de arquivos – normalmente não fornecem o tipo dos serviços exigidos pelo SGBD e sugere algumas melhorias para esses recursos.

D.45 M. Schkolnick: "A Survey of Physical Database Design Methodology and Techniques", Proc. 4th Int. Conf. on Very Large Data Bases, Berlim, Alemanha (setembro de 1978).

D.46 S. Finkelstein, M. Schkolnick e P. Tiberio: "Physical Database Design for Relational Databases", *ACM TODS 13,* Número 1 (março de 1988).

Em alguns aspectos, o problema do projeto de banco de dados físico é mais difícil nos sistemas relacionais do que em outros tipos de sistema. Isso é porque é o sistema, não o usuário, que decide como "navegar" na estrutura de armazenamento; portanto, o sistema terá apenas uma chance de operar bem se as estruturas de armazenamento escolhidas pelo projetista de banco de dados forem adequadas ao que o sistema realmente precisa – o que implica que o projetista precisa saber com alguma profundidade como o sistema funciona internamente. E os projetistas normalmente não terão esse conhecimento (nem é desejável que eles tenham, ou devam ter). Conseqüentemente, algum tipo de ferramenta de projeto físico automatizado é altamente desejável. Este documento oferece informações sobre uma ferramenta desse tipo, chamada DBDSGN, que foi desenvolvida para operar com o System R [4.1–4.3, 4.12–4.14]. O DBDSGN toma como entrada uma definição de workload (ou seja, um conjunto de requisições de usuário e suas freqüências de execução correspondentes) e produz como saída um projeto físico sugerido (ou seja, um conjunto de índices para cada arquivo, geralmente incluindo um "índice de cluster" – veja o Exercício D.6 – em cada caso). Ele interage com o otimizador do sistema (veja o Capítulo 18) para obter informações como o conhecimento que o otimizador tem do banco de dados (com relação aos tamanhos de arquivo, por exemplo) e as fórmulas de custo usadas pelo otimizador.
O DBDSGN foi usado como a base para um produto da IBM chamado RDT, que foi uma ferramenta de projeto para o SQL/DS [4.14].

D.47 Kenneth C. Sevcik: "Data Base System Performance Prediction Using an Analytical Model", Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, França (setembro de 1981).

Como seu título sugere, o objetivo deste artigo é mais abrangente do que o do presente apêndice – ele trata dos problemas de desempenho de sistema em geral, não apenas das estruturas de armazenamento em si. O autor propõe uma estrutura em camadas na qual várias decisões de projeto (e as interações entre essas decisões) podem ser sistematicamente estudadas. As camadas da estrutura representam o sistema como níveis de descrição cada vez mais detalhados; assim, cada camada é mais específica (ou seja, entra em um nível de abstração mais baixo) do que a camada anterior. Os nomes das camadas dão alguma idéia dos níveis de detalhe correspondentes: mundo abstrato, banco de dados lógico, banco de dados físico, acesso de unidade de dados, acesso de E/S físico e carregamentos de dispositivo. O autor afirma que um modelo analítico baseado nessa estrutura poderia ser usado para prever inúmeras características de desempenho, inclusive utilização de dispositivo, throughput de transação e tempos de resposta.
O artigo inclui uma extensa bibliografia comentada sobre o desempenho de sistema. Em especial, ele inclui um breve estudo sobre o desempenho das diferentes estruturas de armazenamento.

D.48 Hanan Samet: "The Quadtree and Related Hierarchical Data Structures", *ACM Comp. Surv. 16,* Número 2 (junho de 1984).

As estruturas de armazenamento descritas no presente apêndice funcionam bem o suficiente para os banco de dados comerciais tradicionais. Porém, à medida que o campo da tecnologia de banco de dados se

expande para incluir novos tipos de dados – por exemplo, dados espaciais, como os que poderiam ser encontrados em aplicações cartográficas ou de processamento de imagens – novos métodos de representação de dados em nível de armazenamento também se tornam necessários. Este documento é uma introdução tutorial a alguns desses novos métodos. Para obter mais informações, veja o livro de Samet [D.49] e as referências [D.50–D.59].

D.49 (O mesmo que a referência [26.37].) Hanan Samet: *The Design and Analysis of Spatial Data Structures*. Reading, Mass.: Addison-Wesley (1990).

Veja a anotação à referência anterior.

D.50 Stavros Christodoulakis e Daniel Alexander Ford: "Retrieval Performance Versus Disc Space Utilization on WORM Optical Discs", Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (maio/junho de 1989).

Veja a anotação à referência [D.51].

D.51 David Lomet e Betty Salzberg: "Access Methods for Multiversion Data", Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. (maio/junho de 1989).

Um *disco WORM* é um disco ótico com a propriedade de que, uma vez que um registro tenha sido gravado no disco, ele nunca pode ser substituído – ou seja, ele não pode ser atualizado no local (WORM é um acrônimo que significa "Write Once, Read Many times" – gravar uma vez, ler muitas vezes). Esses discos apresentam vantagens óbvias para certas aplicações, particularmente as que envolvem algum tipo de necessidade de arquivamento. Porém, as estruturas de armazenamento tradicionais, como árvore-b, não são adequadas a esses discos, precisamente devido ao fato de que a regravação é impossível. Portanto (como explicado na anotação à referência [D.48], embora por questões diferentes), novas estruturas de armazenamento são necessárias. As referências [D.50] e [D.51] propõem e analisam algumas dessas estruturas; a referência [D.51], em especial, trata das estruturas que são apropriadas a aplicações em que um registro histórico completo deve ser mantido – isto é, aplicações em que os dados apenas são incluídos no banco de dados, nunca excluídos (veja o Capítulo 23).

D.52 J. Encarnação e F.-L. Krause (eds.): *File Structures and Data Bases for CAD*. Nova York, N.Y.: North-Holland (1982).

As aplicações de CAD (projeto auxiliado por computador) são uma importante motivação por trás do estudo sobre novas estruturas de armazenamento. Este livro consiste nas atas de um workshop sobre o assunto, cujas seções principais são as seguintes:

1. Modelagem de dados para CAD
2. Modelos de dados para modelagem geométrica
3. Bancos de dados para modelagem geométrica

41. Estruturas de hardware

5. Aspectos do estudo de banco de dados de CAD
6. Problemas de implementação em sistemas de banco de dados de CAD
7. Aplicações industriais

D.53 R. A. Finkel e J. L. Bentley: "Quad-Trees – A Data Structure for Retrieval on Composite Keys", *Acta Informatica 4* (1974).

D.54 J. Nievergelt, H. Hinterberger e K. C. Sevcik: "The Grid File: An Adaptable, Symmetric, Multikey File Structure", *ACM TODS 9,* Número 1 (março de 1984).

D.55 Antonin Guttman: "R-Trees: A Dynamic Index Structure for Spatial Searching", Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. (junho de 1984).

D.56 Nick Roussopoulos e Daniel Leifker: "Direct Spatial Search on Pictorial Databases Using Packed R-Trees", Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. (maio de 1985).

D.57 D. A. Beckley, M. W. Evens e V. K. Raman: "Multikey Retrieval from K-D Trees and Quad-Trees", Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. (maio de 1985).

D.58 Michael Freeston: "The BANG File: A New Kind of Grid File", Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. (maio de 1987).

D.59 Michael F. Barnsley e Alan D. Sloan: "A Better Way to Compress Images", *BYTE 13,* Número 1 (janeiro de 1988).

Descreve uma nova técnica de compactação para imagens, chamada *compactação fractal*. A técnica funciona armazenando não a imagem em si, mas os valores de código que podem ser usados para recriar a imagem desejada como e quando necessário através de equações fractais apropriadas. Desse modo, podem ser conseguidas "taxas de compactação de 10.000 para 1 – ou ainda mais altas".