Unidade 5 - Organização de dados em memória secundária

5.1 - Primeiras palavras

Em contraposição à memória principal, dados armazenados em memória secundária sofrem os problemas de latência intrínsecos aos dispositivos externos. No caso de um disco rígido, estas restrições são mecânicas e o tempo de acesso aos dados depende da movimentação do braço de leitura e escrita, além da própria rotação do disco e transferência de dados para a memória principal.

No caso de armazenamento em memória secundária, portanto, uma das principais preocupações é a redução do número de acesso a disco. Quando um dado precisa ser recuperado do disco, o acesso é por blocos, o que significa que, na maioria dos casos, vários registros são recuperados de uma vez. Aproveitar esta característica de acesso é importante para que não sejam necessários mais acessos a disco que os realmente necessários para executar uma dada operação.

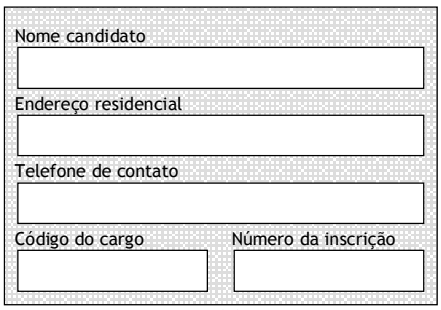
É também importante lembrar que o tempo de acesso a disco é muito maior que o tempo de acesso à memória principal. Assim, mesmo nas situações em que se precise trabalhar bastante os dados que estão em memória, o tempo deste processamento tende a ser bem menor que o desprendido para trazer os dados do disco para a memória.

Esta unidade apresenta algumas das principais formas de armazenamento de dados em memória secundária. Em princípio, a mesma visão adotada para a organização de dados em memória principal é seguida. Porém o fato dos dados estarem em disco será considerado e as consequências deste fato serão discutidas.

5.2 - Tabelas em memória secundária e restrições de acesso

O ponto de partida para entender as limitações do uso de memória secundária é rever alguns conceitos importantes envolvidos. Inicialmente, é importante que se entendam as relações entre os **registros** e os **arquivos**, para posteriormente vincular a estes conceitos o acesso a disco, caracterizado pelo uso de blocos para acesso de leitura e gravação.

Um **registro** é uma unidade de armazenamento. Em algoritmos, os registros também são chamados de variáveis compostas heterogêneas, dada sua natureza de armazenar um conjunto de informações cujos tipos podem ser diferentes uns dos outros. A Figura 5-1 ilustra um registro com informações básicas sobre um candidato a um emprego de uma empresa. É possível notar que dois tipos de dados são utilizados (**literal** e **inteiro**), compondo os cinco **campos** necessários.



(a)

```
tipo tCanditado:
          registro
            nomeCandidato,
            endereçoResidencial,
            telefoneContato: literal
            códigoCargo,
            númeroInscrição: inteiro
          fim-registro
```

(b)

Figura 5-1. Exemplo de um registro: (a) Esquema representativo das informações; (b) declaração do tipo de dados.

Registros podem ser utilizados, por exemplo, como variáveis individuais em um algoritmo ou um programa, quando são úteis como organização de dados, o que auxilia na clareza e documentação. O aumento do volume de dados pode ser conseguido também com o uso de vetores de registros (tabelas), também ainda no contexto da memória principal.

Quando se deseja guardar um grande número de registros de forma permanente, o uso da memória secundária se faz necessário. Assim, coleções de registros podem ser guardadas em **arquivos**. Um arquivo pode ser visto, do ponto de vista conceitual, como uma sequência de dados; em particular, no enfoque desta discussão, cada dado individual corresponde a um registro. Um arquivo, então, pode ser considerado como uma coleção de registros, cada qual ocupando uma posição. Por convenção, a numeração das posições se inicia em zero. Um exemplo de um arquivo com quatro registros é apresentado na Figura 5-2.

O ponto essencial para que se entendam os arquivos como modo de armazenamento é lembrar que estão em memória secundária. Isso significa que não estão diretamente disponíveis para uso em algoritmos ou programas em execução. Para que cada registro fique disponível no contexto do programa, é preciso que seja feita uma operação de leitura, o que implica no acionamento, por parte do sistema operacional,

das ações para que os bytes armazenados nos setores de um disco rígido sejam transferidos para a memória principal disponível para o programa, passando pelo buffer do sistema.

Figura 5-2. Esquema mostrando um arquivo com quatro registros, ocupando as posições de 0 a 3.

Outros destaques em relação aos arquivos compreendem:

- Diferentemente de tabelas em memória, arquivos não possuem um índice que permite acesso a cada registro. Cada registro deve ser especificado por sua posição e uma operação de leitura deve ser realizada para copiar os dados (bytes) do disco rígido para uma variável na memória principal.
- Os campos não ficam disponíveis separadamente. Usualmente, é necessário que um registro completo seja lido para a memória principal para, então, um de seus campos poder ser utilizado.[*]
- Alterações de informações exigem leitura do dado para uma variável em memória, modificação desta variável e posterior armazenamento (gravação) do registro de volta no arquivo, sobrescrevendo a informação ali armazenada.
- O tamanho dos arquivos é dinâmico. Quando criado possui tamanho zero, com nenhum registro. À medida que operações de gravação são feitas no final do arquivo, seu tamanho aumenta. As limitações para o tamanho de um arquivo são dadas pelo sistema operacional e pelo espaço disponível no disco rígido.
- Arquivos não podem ser reduzidos a não ser por **truncamento**, que é uma operação que descarta a parte final do arquivo. Isto significa que não é possível remover do arquivo uma parte intermediária, mas somente a final. Esta operação consiste, em termos práticos, em marcar um dos registros e descartar do arquivo ele e todos seus subsequentes.

Todas estas características aplicam-se aos aspectos **lógicos** de funcionamento de arquivos. Em outras palavras, a visão de um arquivo linear é uma abstração. Quando ligado à parte física do armazenamento de dados em discos rígidos, outros aspectos devem ser considerados. Alguns destes são

[*] Recuperação de partes de registros não é uma operação comum. Embora seja possível ser feita, a complexidade do controle necessário para desenvolver um algoritmo nestes moldes aumenta significativamente, sem que se obtenha vantagem significativa com isso.

menos significativos e serão apenas comentados ou até ignorados, outros são importantes e terão o destaque necessário.

O sistema operacional é responsável por mapear um arquivo nos setores (e trilhas e cilindros, por consequência) do disco rígido. Ao fazer isso, toma decisões sobre onde e quando realizar as gravações. Assim, embora um arquivo seja uma estrutura contínua do ponto de vista lógico, o sistema operacional pode mapear vários pedaços dos dados em partes não necessariamente consecutivas do disco[*]. O resultado deste comportamento é o aumento do tempo necessário para leituras posteriores, já que latências adicionais de busca e rotação são envolvidas. Como não há controle do desenvolvedor dos programas sobre este comportamento, é importante saber que ele existe e quais suas consequências. Para as considerações de desempenho neste texto não serão consideradas estas especificidades.

O acesso para leitura ou gravação de informações em discos rígidos, dadas as diversas latências envolvidas, possui, como se sabe, tempo de recuperação e armazenamento muito superiores quando comparados a acessos em memória principal. Esta questão é fundamental quando são considerados registros armazenados em arquivos. O uso de estratégias que considerem a busca de dados em blocos são importantes, visto que pode ser minimizado o número de acessos a disco e, em consequência, o tempo gasto pela execução das operações de manipulação de dados.

Esta unidade aborda as organizações de arquivos segundo algumas visões importantes, em ordem crescente de complexidade. Para as discussões, são sempre considerados registros de estrutura e tamanho fixos[†], de forma que a localização de um registro dento dos bytes que compões um arquivo seja possível.

Em um primeiro momento, são considerados os arquivos sem ordenação e, em seguida, os arquivos ordenados. Estas duas formas de organização, consideradas sob o ponto de vista do conjunto de operações possíveis sobre tabelas, permitem expandir o conceito para uma organização mais sofisticada, que envolve os arquivos indexados.

5.3 - Formas de organização

Quando coleções de registros são mantidas em arquivos são obtidas estruturas chamadas de **tabelas em arquivo**, ou simplesmente **tabelas**. Embora o termo seja usado tanto para as estruturas mantidas em memória principal quanto secundária, o contexto permite diferenciá-las quanto a estrutura e manipulação, mas principalmente quanto às formas de acesso. O ponto mais importante de diferenciação, entretanto, é o tempo de acesso à memória secundária ser muito superior quando comparado à memória principal.

As diversas operações sobre tabelas serão trabalhadas para arquivos sem ordenação, caracterizando os acessos a disco frente às manipulações de dados, firmando os conceitos envolvidos. Em seguida as modificações necessárias para a organização de arquivos ordenados será abordada.

5.3.1 - Arquivos sem ordenação

Um arquivo sem ordenação corresponde a um arquivo cujos registros não possuem necessariamente qualquer forma de posicionamento dos registros um em relação ao outro em função do seu conteúdo.

Assumindo que se disponha de um conjunto inicial de dados que devem ser usados para formar o arquivo inicial, a **criação** do arquivo é feita pela criação de um arquivo vazio, seguida da escrita dos registros, um por vez, no final do arquivo, fazendo com que este último aumente de tamanho a cada novo registro. Considerando-se que o tamanho do problema seja o número de registros inicialmente disponíveis, então a complexidade da criação é O(*n*).

Do ponto de vista conceitual, a operação é simples e consiste em um laço de repetição que faz as diversas escritas no arquivo. A estrutura essencial da lógica de inserção é apresentada no Algoritmo 5-1.

Algoritmo 5-1

```
1  { criação do arquivo não ordenado }
2  associeArquivo(arquivoDados, "nome_arquivo.dat")
3  crieArquivo(arquivoDados)  { cria arquivo vazio }
4
```

[*] Neste caso, o termo usado é **fragmentação** do arquivo.

[†] Os registros mais usuais são registros fixos tanto nos campos quanto no tamanho de cada campo. Porém, é possível que sejam definidos e manipulados registros de tamanhos variáveis, o que envolve registros que podem ou não possuir determinados campos (exemplo: se um campo com o número de filhos for nulo, não é preciso ter campos para o nome dos filhos) ou um mesmo campo ocupar mais ou menos espaço para registros diferentes (exemplo: nomes mais longos ocupam mais bytes que nomes curtos). Ou ainda ambos os casos acontecerem simultaneamente. Nestas situações, os controles adicionais sobre o conteúdo do registro e as posições das informações internamente a ele são necessárias. Estas informações são conhecidas por **metadados** e podem existir para o registro, para o arquivo ou para ambos.

```
5  leia(númeroInicialRegistros)
6  para i ← 1 até númeroInicialRegistros faça
7       obtenhaRegistro(registro) { digitação dos dados por exemplo }
8       escrevaArquivo(arquivoDados, registro) { escreve novo registro }
9  fim-para
10
11 fecheArquivo(arquivoDados) { encerra o acesso }
```

Há que se considerar, porém, ainda os aspectos físicos para a criação do arquivo. Cada comando de escrita, antes de provocar o armazenamento efetivo dos dados em disco, faz uma transferência do registro escrito para o buffer interno. Somente quando todos os bytes do buffer são preenchidos é que a escrita em disco tem efeito. Assim, supondo que em um bloco de disco caibam exatos 50 registros, somente após todos estarem presentes no buffer é que o sistema operacional determina um bloco no disco e faz a transferência, em uma única operação, de todo o conteúdo do buffer.

É, ainda, possível (e provável) que em um buffer não caiba um número exato de registros. Nestes casos, os primeiros bytes do registro ficam contidos em um bloco, enquanto os restantes são alocados no bloco seguinte. Durante as operações de leitura de dados, o sistema operacional se incumbe de "montar" o registro de volta, agrupando novamente os bytes. Nestes casos, a leitura de um simples registro pode exigir dois acessos a disco.

No caso da criação do arquivo, porém, a quebra de um registro em blocos diferentes não é um problema, visto que cada bloco é preenchido e gravado uma única vez, gerando o mínimo de acessos a disco possível.

Os comandos de fechamento de arquivo liberam o sistema operacional de controlar o arquivo, o que faz com que o último bloco, mesmo incompleto, seja escrito efetivamente no disco. Este último bloco pode ser composto parte por registros válidos e parte por bytes quaisquer, que já estavam no buffer. Todos estes bytes são escritos no disco, mesmo o "lixo" ali presente. Como o sistema operacional mantém o controle sobre o número total de bytes válidos, os bytes extras são facilmente ignorados e descartados.

A **inserção** de um novo registro ao arquivo segue lógica similar à da criação, com a escrita do novo item na última posição. Assumindo que o arquivo já esteja aberto para acesso para leitura e gravação, o Algoritmo 5-2 apresenta os passos para uma nova inclusão.

Algoritmo 5-2

```
1  insira(arquivoDados, novoRegistro)
2       { posicionamento no fim do arquivo }
3       posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados))
4
5       { gravação do novo item }
6       escrevaArquivo(arquivoDados, novoRegistro)
```

No Algoritmo 5-2, considera-se que a função **tamanhoArquivo** retorne o número de registros; como os registros existentes estão nas posições de 0 a **tamanhoArquivo(arquivoDados)** – 1, o comando **posicioneArquivo** da linha 3 ajusta o arquivo para escrever após a última posição. Assim, o arquivo é aumentado em um registro, consumindo tempo O(1).

Quando se consideram os acessos efetivos ao disco rígido, é preciso lembrar que todo o bloco que vai conter o novo registro tem que ser reescrito no disco. Para tanto, é preciso, inicialmente, que o bloco seja lido para o buffer e manipulado para que passe a conter o novo item. Então o bloco todo é reescrito de volta no disco. Assim, são necessários dois acessos a disco para que a inserção possa ser feita.

Um ponto importante nesta discussão é que o sistema operacional é o responsável pela leitura e escrita dos dados do buffer. Portanto, uma leitura somente será necessária se o bloco ainda não estiver presente no buffer. Da mesma forma, o sistema decide quando o bloco será reescrito no disco, o que pode levar à situação em que várias novas inserções podem ser feitas aproveitando o bloco já lido para o buffer, poupando escritas em disco. Não há como prever, do ponto de vista do desenvolvedor do algoritmo, quando as escritas efetivamente ocorrem, mas que pelo menos uma leitura e uma escrita ocorrem é fato.

O procedimento de **remoção** em arquivos não ordenados segue um raciocínio simples para manter os dados do arquivo organizados: o registro da última posição é copiado sobre o registro a ser apagado e, em seguida, o arquivo é truncado de forma a perder a cópia do registro recém-copiado. Esta operação tem complexidade O(1).

Algoritmo 5-3

```
1  remova(arquivoDados, posição)
2       { obtenção do último registro }
3       posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados) – 1)
4       leiaArquivo(arquivoDados, últimoRegistro)
5
6       { gravação do registro sobre a posição
7         do registro a ser descartado }
```

```
8       posicioneArquivo(arquivoDados, posição)
9       escrevaArquivo(arquivoDados, últimoRegistro)
10
11      { truncamento do arquivo }
12      posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados))
13      trunqueArquivo(arquivoDados)
```

Em termos de acesso a disco, esta operação exige a leitura do último bloco do arquivo, onde está o último registro, e também a leitura do bloco em que está o registro a ser removido (caso não sejam o mesmo). Com os blocos disponíveis, a cópia do último registro sobre o removido é feita e o bloco deve ser reescrito em disco. Finalmente, para o truncamento*, novo acesso a disco deve ser feito para atualizar o último bloco, agora sem o registro final.

A **pesquisa** em arquivos não ordenados é feita de forma sequencial, até localizar o registro ou chegar ao final do arquivo, sendo esta última a situação de falha na busca. A busca é feita pela leitura dos registros na ordem de armazenamento, um a um, como ilustrado pelo Algoritmo 5-4.

Algoritmo 5-4

```
1   pesquisa(arquivoDados, chave)
2       { ajusta busca para o início do arquivo }
3       posiçãoAnterior ← posiçãoArquivo(arquivoDados) { preserva posição atual }
4       posicioneArquivo(arquivoDados, 0)
5
6       achou ← falso
7       posição ← -1 { contagem atrasada }
8       enquanto não achou e não fimArquivo(arquivoDados) faça
9             leiaArquivo(arquivoDados, registro)
10            achou ← registro.chave = chave
11            posição ← posição + 1
12      fim-enquanto
13
14      { restaura posição anterior do ponteiro do arquivo }
15      posicioneArquivo(arquivoDados, posiçãoAnterior)
16
17      { retorna resultado da pesquisa }
18      se achou então
19            retorne posição
20      senão
21            retorne -1 { indicação de falha }
22      fim-se
```

Em termos de acesso a disco, a busca lê tantos blocos quanto necessários de forma sequencial, examinando os registros contidos em cada um deles. Assumindo que todos os registros tenham a mesma probabilidade de serem pesquisados, em média uma busca bem sucedida varre metade dos registros do arquivo, o que também corresponde à metade dos blocos que formam o arquivo. No caso de busca por um registro não existente, todo o arquivo é analisado e todos seus blocos lidos para o buffer. Em qualquer das situações, o tempo é proporcional a O(*n*), sendo *n* o número de registros presentes no arquivo.

As **atualizações** são feitas de forma direta, com a leitura e reescrita do registro no mesmo local, como apresentado no Algoritmo 5-5, o qual assume que a alteração seja feita de alguma forma no procedimento **editaRegistro**. Esta operação é O(1).

Algoritmo 5-5

```
1   altere(arquivoDados, posição)
2       { recuperação do registro }
3       posicioneArquivo(arquivoDados, posição)
4       leiaArquivo(arquivoDados, registro)
5
6       { modificação dos dados }
7       editaRegistro(registro)
8
9       { regravação do registro }
10      posicioneArquivo(arquivoDados, posição)
11      escrevaArquivo(arquivoDados, registro)
```

No caso de atualizações, o bloco com o registro tem que ser lido do disco, os bytes referentes ao registro modificado têm que ser atualizados e, então, o bloco do buffer reescrito de volta no disco. Assim, dois acessos a disco são necessários, caso os dados não estejam disponíveis no buffer e para garantir que a efetiva escrita no disco seja completada.

* A operação de truncamento, em muitas linguagens, pode não poder ser feita com o arquivo aberto. No caso, o arquivo tem que ser fechado, truncado e, então, reaberto. Estas operações consomem tempo e, possivelmente, muitos acessos a disco. Estes não serão considerados neste texto.

Arquivos não ordenados, estruturados da forma descrita, não requerem **manutenção** e, assim, mais nenhuma ação é necessária para definir as operações de manipulação de dados.

As operações descritas referem-se, em termos das considerações sobre necessidades de leitura e escrita dos blocos de disco, a registros que estejam integralmente contidos em um único bloco. Conforme já comentado, é possível que alguns registros fiquem parte em um bloco, parte em outro. Nestes casos, ambos os blocos precisam ser lidos e atualizados, o que pode gerar, para estas situações específicas, tempo de acesso maior que para os demais registros.

Finalmente, uma consideração adicional ainda deve ser feita. Da mesma forma que em tabelas em memória, as remoções em arquivos não ordenados podem ser feitas pela simples marcação de um campo de validade do registro. Esta alternativa exige leitura e regravação apenas do bloco que contém o registro removido logicamente. Como outras implicações, a pesquisa deve ignorar os registros inválidos e uma operação de manutenção, que usualmente é feita pela cópia dos registros válidos para um novo arquivo (eliminando assim os marcados como inválidos), e colocando este novo arquivo no lugar do anterior.

5.3.2 - Arquivos ordenados

Arquivos ordenados são, assim como os não ordenados, conjuntos de registros armazenados sequencialmente em memória secundária. Há, porém, a restrição de que a posição relativa dos registros obedeça a um critério de ordem, que é aplicado a uma **chave de ordenação**. Como consequência, as operações sobre as tabelas em disco devem prever que esta ordenação seja mantida e também usufruir das vantagens dela, que é o caso da pesquisa, que pode usar o algoritmo de busca binária. Convém ainda lembrar que a ordenação é apenas para uma chave específica e pesquisas por outras chaves que não a de ordenação seguem a operação sobre arquivos não ordenados.

Considerando um conjunto inicial de dados, um arquivo ordenado é **criado** pela colocação de todos os registros em um arquivo sem ordenação e, então, submetendo este arquivo a uma ordenação apropriada, como é o caso do mergesort, procurando mesclar ordenações parciais em memória principal (que são rápidas) com a fusão destas sequências ordenadas até que se obtenha o arquivo completo ordenado.

Algoritmo 5-6

```
1  { criação do arquivo ordenado }
2  associeArquivo(arquivoDados, "nome_arquivo.dat")
3  crieArquivo(arquivoDados)  { cria arquivo vazio }
4
5  leia(númeroInicialRegistros)
6  para i ← 1 até númeroInicialRegistros faça
7       obtenhaRegistro(registro) { digitação dos dados por exemplo }
8       escrevaArquivo(arquivoDados, registro)
9  fim-para
10
11 { ordenação }
12 mergesort(arquivoDados)
13
14 fecheArquivo(arquivoDados) { encerra o acesso }
```

Os acessos a disco devem ser minimizados para que a operação consuma menos tempo. Para isso, além do uso da memória principal para ordenações de parte dos dados, o mergesort varre cada partição ordenada do início ao fim, de forma que cada registro (e, portanto, cada bloco) seja lido apenas uma vez a cada passo. A complexidade de tempo desta etapa é vinculada à complexidade do algoritmo de ordenação.

Inserções de novos registros requerem a manutenção da ordenação. Na prática, isso significa que os registros devem ser "movidos" o suficiente para que o novo item, ao ser escrito, ocupe sua posição relativa correta na ordenação. Cada registro, iniciando-se no último, que tiver sua chave de ordenação maior que a do novo registro, deve ser movido uma posição para frente. Uma movimentação de registro significa sua leitura de um local e seu armazenamento na posição subsequente.

O Algoritmo 5-7 apresenta a movimentação dos registros com chave maior que a do novo registro e a consequente escrita do novo registro no local correto.

Algoritmo 5-7

```
1  insira(arquivoDados, novoRegistro)
2       { movimentação dos registros necessários para manter a ordem }
3       posição ← tamanhoArquivo(arquivoDados) – 1 { último }
4
5       terminou ← posição < 0
6       enquanto não terminou faça
7             posicioneArquivo(arquivoDados, posição)
8             leiaRegistro(arquivoDados, registro)
```

```
9
10            se registro.chave > novoRegistro.chave então
11                  escrevaArquivo(arquivoDados, registro) { reescreve na próxima posição }
12                  posição ← posição - 1
13            senão
14                  terminou ← verdadeiro
15            fim-se
16      fim-enquanto
17
18      { escreve novo registro na posição correta }
19      escrevaArquivo(arquivoDados, novoRegistro)
```

Supondo que não haja qualquer tendência quanto aos valores das chaves dos novos registros inseridos, é possível pensar que, em média, metade dos registros é movida para uma inserção. O custo computacional da operação é proporcional a $n/2$ e, assim, $O(n)$.

Quando o último registro é lido, todo o bloco que o contém é transferido para a memória. Quando as leituras e gravações para a movimentação dos registros ocorrem em um mesmo bloco, não há necessidade de que ele seja escrito no disco a todo o momento. É razoável imaginar que as movimentações ocorram todas internamente e, quando o bloco estiver correto, ele é reescrito modificado no disco rígido. Para que um bloco fique correto após as movimentações, é preciso que um registro (ou parte de um registro) do bloco anterior seja gravado nele. Esta situação leva à leitura do bloco anterior, seguida da movimentação do registro e, então, possibilitando a gravação do bloco. O mesmo procedimento ocorre para o todos os blocos, do fim para o início do arquivo, até que se localize a posição correta para inserção. Em média, metade dos blocos do arquivo precisa ser reescrita.

Uma **remoção** de um registro de uma dada posição, similarmente à inserção, requer a movimentação dos registros. Depois o último registro é eliminado por truncamento, pois ele fica duplicado. Assumindo que todo registro tenha a mesma chance de ser removido, em média metade dos registros tem que ser reescrita e a operação, assim, assume tempo $O(n)$.

Algoritmo 5-8

```
1   remova(arquivoDados, posição)
2       { movimentação dos registros subsequentes }
3       posiçãoÚltimo ← tamanhoArquivo(arquivoDados) - 1
4       para posiçãoAtual ← posição + 1 até posiçãoÚltimo faça
5             { obtém o registro }
6             posicioneArquivo(arquivoDados, posiçãoAtual)
7             leiaArquivo(arquivoDados, registro)
8
9             { reescreve na posição anterior }
10            posicioneArquivo(arquivoDados, posiçãoAtual - 1)
11            escrevaArquivo(registro)
12      fim-para
13
14      { truncamento do arquivo }
15      posicioneArquivo(arquivoDados, tamanhoArquivo(arquivoDados)
16      trunqueArquivo(arquivoDados)
```

Ao se considerar o acesso a disco usando os blocos, é preciso lembrar que um bloco é lido para o buffer, as movimentações de registro internas são realizadas e o bloco é regravado em disco. O detalhe de um registro (ou parte dele) ter que ser movido entre blocos também ocorre nesta situação. A regravação do último bloco devido ao truncamento é também necessária, embora possa certamente ocorrer juntamente com as movimentações feitas neste último bloco.

A **pesquisa** em arquivos ordenados pode empregar a pesquisa binária, cuja ordem de complexidade de tempo é $O(\log n)$. O algoritmo da pesquisa binária não é apresentado aqui e difere da implementação convencional apenas pela necessidade de, ao se calcular a posição do meio, o registro tem que ser lido antes de se tomar a decisão sobre em qual metade do arquivo a busca deve continuar. A questão dos blocos lidos do disco, por sua vez, merece alguns comentários. Na primeira leitura (registro do meio do arquivo), todo o bloco que o contém é lido para o buffer e, caso este registro esteja dividido em blocos diferentes, ambos devem ser lidos. A cada passo da repetição da pesquisa binária, novo posicionamento e leitura de registro são necessários e, novamente, todo o bloco é lido para recuperar um único registro. Ao final da pesquisa, quando os "saltos" da busca se tornam menores, é provável que a busca acabe sendo feita dentro de um único bloco, o que evita novos acessos a disco. Apesar destas considerações, a pesquisa binária ainda é muito eficiente e o custo da manutenção de um arquivo ordenado é recompensada por uma pesquisa muito eficiente.

A **alteração** dos dados de um registro em um arquivo ordenado pode gerar a necessidade de reorganização, o que acontece se a chave de ordenação é o valor modificado. Neste caso, os deslocamentos são mais uma vez necessários para colocar o registro na posição correta. Em termos práticos, o algoritmo de movimentação deve verificar se a movimentação deve acontecer em direção ao

início ou ao fim do arquivo e, um a um, ir movendo os registros até que seja possível colocar o registro modificado em sua nova posição. Considerando que a modificação da chave possa ser uma modificação qualquer, é possível assumir que, uma quantidade de registros proporcional a *n* deva ser movida em cada alteração, com complexidade resultante O(*n*). Finalmente, as discussões feitas sobre o uso dos blocos para a inserção e remoção também se aplicam a esta operação.

Já que todas as operações mantêm a tabela ordenada o tempo todo, não se requer a operação de **manutenção**.

5.3.3 - Arquivos com controle dos blocos

Das considerações feitas no tópico anterior, é possível notar a possibilidade de que um registro fique dividido entre dois blocos, o que requer, para estes registros, acessos adicionais tanto para leitura quanto para gravação. Por exemplo, se um bloco em disco rígido possuir 4096 bytes (4 KiB) e cada registro tiver 100 bytes, então caberão em um primeiro bloco 409 registros inteiros. O próximo registro terá seus 6 primeiros bytes armazenados no primeiro bloco e os restantes 94 no segundo. O segundo bloco terá ainda 400 registros inteiros e mais 2 bytes iniciais de um registro que será dividido entre o segundo e terceiro blocos. E assim sucessivamente.

O gerenciamento da utilização do buffer e dos blocos pelo sistema operacional pode, entretanto, ser controlado pelo programador, de forma que uma utilização mais racional do conhecimento dos blocos evite quebras de registro, por exemplo.

Este tópico mostra uma forma de realizar este controle e os conceitos aqui apresentados serão recuperados nas discussões sobre arquivos indexados.

Como cenário, considera-se um bloco de tamanho *b* bytes e registros com *r* bytes cada um, sendo *r* menor do que *b*. Calculando-se o maior inteiro resultante da razão *b*/*r* obtém-se o número de registros inteiros que podem ser armazenados em um bloco. A este valor é representado por *k*. Por fim, ao se calcular *l* = *b* – *kr* consegue-se o número de bytes restantes dentro do bloco. A Figura 5-3 apresenta uma visão geral sobre o uso do bloco.

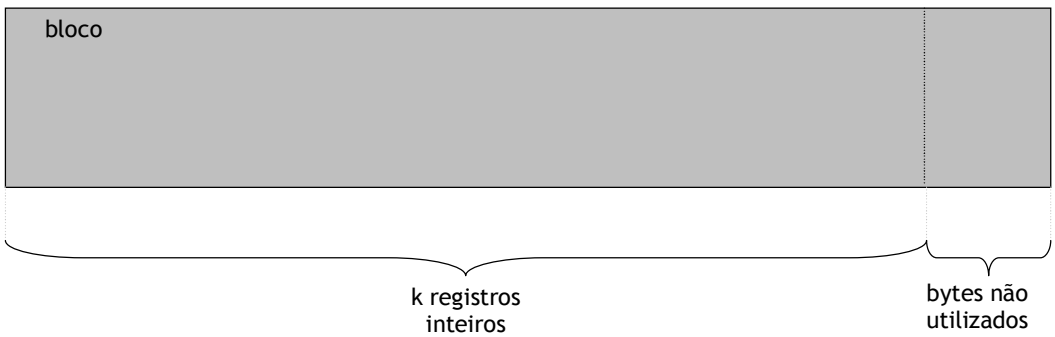


Figura 5-3. Ilustração de um bloco ocupado parte por registros inteiros e parte não utilizado.

Uma opção para controlar os blocos é definir um arquivo cujos registros tenham exatamente o tamanho de um bloco. Para este arquivo "especial", segundo o cenário apresentado, cada registro seria composto de um vetor com *k* registros de dados, acrescido de um contador para indicar, dentro do bloco, quantos registros de dados são válidos e um vetor não utilizado, criado apenas para garantir que o número de bytes corresponda ao tamanho de um bloco. O Algoritmo 5-9 mostra uma possível declaração para o registro de bloco.

Algoritmo 5-9

```
1  { registro de controle de bloco }
2  tipo tBloco: registro
3                    dados[k]: tRegistroDados
4                    contador: inteiro
5                    lastro[l]: byte
6              fim-registro
```

A Figura 5-4 mostra como o registro pode ser usado para armazenar dados. No exemplo, assume-se que cada bloco seja capaz de armazenar 7 registros inteiro (*k*), usando o vetor **dados**. Os *l* bytes restantes no bloco são usados para manter o campo **contador**, além do campo **lastro**, que deve ser dimensionado de forma que o número total de bytes do registro de bloco tenha o tamanho de um bloco em disco rígido. Na Figura 5-4 estão armazenados 25 registros de dados, ocupando quatro blocos. O último bloco, ocupado parcialmente, embora armazene efetivamente 7 registros, considera apenas os 4 primeiros, informação dada pelo campo **contador**.

Fazendo-se com que o registro de bloco tenha o tamanho do bloco do disco rígido garante que cada leitura e escrita seja um acesso a disco, nem mais nem menos.

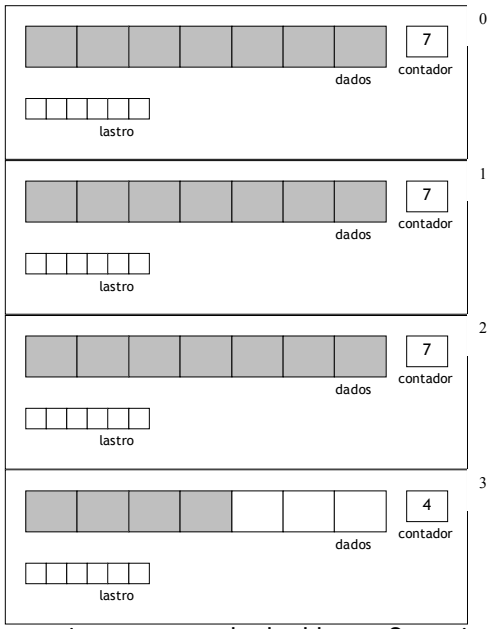


Figura 5-4. Ilustração de um arquivo com controle dos blocos. O arquivo contém 25 registros de dados distribuídos em 4 blocos. O último bloco é ocupado apenas parcialmente. As posições hachuradas indicam registros de dados válidos.

Se este controle for usado para armazenar um arquivo não ordenado, por exemplo, a inserção de um novo registro deve ser feita lendo-se o último registro de bloco, colocando o novo registro de dados na quinta posição do vetor e incrementando o contador para 5 para, finalmente, regravar o bloco de volta.

Cada comando de escrita ou leitura de um algoritmo teria que ser adaptado a esta estrutura. Estes comandos formariam uma interface para um sistema interno que, conhecendo o tipo da organização do arquivo, decidiria sobre o controle dos dados dentro do registro de bloco e quando haveria necessidade de gravação ou leitura dos blocos do disco.

Usando a Figura 5-4 como ilustração e supondo um arquivo sem ordenação, uma remoção do registro da posição 8 geraria a seguinte cadeia de eventos:

1. Leitura do registro de bloco 3 para obtenção do último registro (posição interna 3).
2. Determinação da localização da bloco da posição 8, o que é feito sabendo-se que há 7 registros em cada bloco; o resultado é que tal registro é o da posição interna 1 do bloco 1.
3. Leitura do registro de bloco 1 para acesso ao registro desejado.
4. Cópia do conteúdo do registro da posição interna 3 do bloco 3 para a posição interna 1 do bloco 1.
5. Atualização do contador do bloco 3, com decremento do contador.
6. Gravação dos registros de bloco 1 e 3 para atualizar o arquivo em disco.

Outras operações, envolvendo arquivos não ordenados e ordenados devem refletir esta nova estrutura de organização interna e prover os mecanismos adequados para leituras e gravações. Cada operação sobre a tabela em memória secundária, considerando os registros de bloco, passa a se adequar ao novo modelo.

A opção por controlar os blocos é mais complexa, mas permite maior flexibilidade no uso dos arquivos, aumentando as opções de controle restritas do sistema operacional[*]. Uma vez implementada, praticamente quaisquer arquivos que obedeçam a uma mesma forma de organização podem ser utilizados.

5.3.4 - Arquivos indexados

Um **índice** é uma estrutura auxiliar, mantida paralelamente ao arquivo de dados, que permite localizar com maior eficiência uma dada informação. Nos termos discutidos nesta unidade, um índice é um arquivo separado, cuja função é auxiliar a localização de um dado registro no arquivo de dados.

[*] Na realidade, muitos sistemas de gerenciamento de bancos de dados chegam a eliminar completamente o sistema operacional do processo e manter todas as rotinas de acesso a disco internamente. Obtém, assim, um produto diferenciado no mercado, com desempenho e flexibilidade.

Inicialmente é preciso conhecer o que é o índice e quais informações contém. Dada sua função de auxiliar a localização, uma **entrada de índice** corresponde a uma chave (critério usado para a busca) e a sua localização no arquivo de dados. Os índices são mantidos ordenados (física ou logicamente) para permitir pesquisas eficientes.

Por exemplo, supondo um arquivo com dados sobre alunos e assumindo que a chave usada na pesquisa seja o RA, se um registro que contém os dados sobre o aluno de RA 497728 estiver na posição 8135 do arquivo, então o arquivo de índice terá um registro contendo o par <497728;8135>. O segundo elemento da entrada de índice é chamando **ponteiro** para o registro de dados. Uma pesquisa é feita no arquivo de índice até que se localize o registro desejado. Com isso, é obtido o ponteiro que dá a localização dos dados efetivos no arquivo de dados. A Figura 5-5 ilustra esta relação.

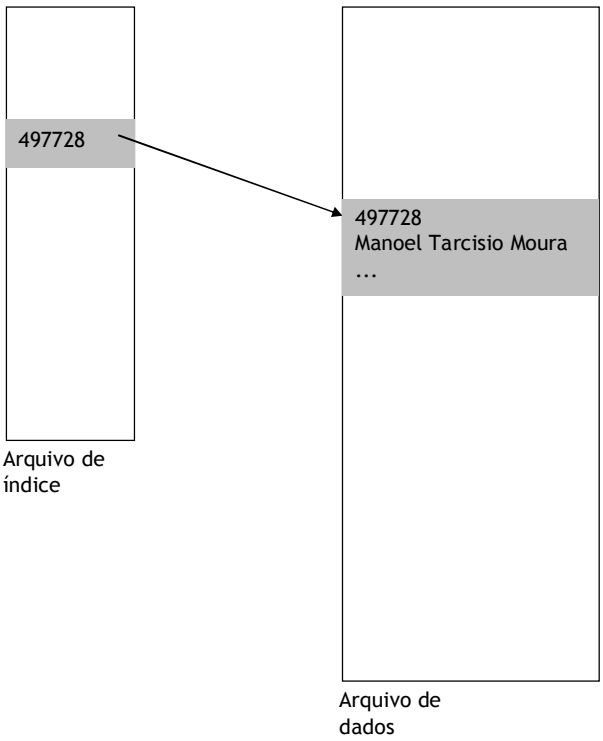


Figura 5-5. Esquema representando, à esquerda, um arquivo de índice e, à direita, o arquivo de dados. Uma entrada de índice para o RA 497728 está em destaque, juntamente com o registro de dados do aluno. A seta representa a indicação do endereço em que o registro de dados está posicionado. As demais entradas de índice e registros são omitidos na figura.

Conhecer as vantagens e desvantagens do uso dos índices é importante. Uma desvantagem evidente é a necessidade de mais espaço em disco para manter a estrutura. Outra desvantagem é que, para que as pesquisas no índice sejam eficientes, é preciso que ele esteja ordenado e isso acarreta tempo para manter a organização do próprio índice.

As vantagens, por seu lado, ficarão claras ao longo do texto, à medida que forem sendo descritos os usos dos índices e suas características.

Como primeiro exemplo será apresentado uma suposição sobre um arquivo com um índice associado a ele. Esta estrutura não é uma organização viável (nem muito prática) e tem a função única de mostrar que índices, por si só, já representam uma vantagem para a pesquisa. Assim, considera-se a seguinte situação:

1. O arquivo de dados é formado por registros com 256 bytes cada um (o que já inclui a chave usada na pesquisa).
2. O arquivo de índice usa uma chave para pesquisa com 10 bytes, além do ponteiro que possui 6 bytes, em um total de 16 bytes por entrada de índice.
3. Todos os registros do arquivo de dados possuem sua respectiva entrada no arquivo de índice, o que leva ambos os arquivos a possuírem exatamente o mesmo número de registros.
4. Ambos os arquivos estão ordenados pela chave usada para pesquisa, o que faz com que a sequência de chaves em um seja idêntica a sequência de chaves no outro.*
5. O bloco de disco em questão possui 2048 bytes.

* Uma pesquisa binária para achar a chave *C* no arquivo de índice, por exemplo, segue exatamente os mesmos passos quando aplicada para achar a mesma chave *C* no arquivo de dados.

6. Não é feito nenhum controle de blocos explícito, deixando as leituras e escritas para o sistema operacional.
7. Cada arquivo contém 25000 registros.

Dadas estas considerações, é possível mostrar que o tempo necessário para localizar uma dada chave no arquivo de índice é menor que a localização da mesma chave no arquivo de dados. Para tanto, basta fazer alguns cálculos. No arquivo de dados, em cada bloco do disco rígido cabem exatos 8 registros de dados. Como são 20.000 registros no arquivo, são consumidos exatos 2500 blocos em disco. Para o arquivo de índice, cada registro ocupa 16 bytes, de forma que cabem 128 entradas de índice em cada bloco. Para 20.000 entradas de índice são consumido, assim, 157 blocos (156 blocos completos e mais um ocupado 25%).

Supondo, agora, que a pesquisa seja sequencial (isto é, um a um a partir do início do arquivo) e que a chave procurada não seja encontrada, o que exigiria que todos os registros tivessem que ser consultados. Nestas condições, a pesquisa no arquivo de índice precisaria de 156 acessos a disco, enquanto a pesquisa no arquivo de dados exigiria 2500 acessos, um valor 16 vezes maior. Mesmo que a pesquisa binária fosse utilizada, o fato de cada acesso a disco disponibilizar um número maior de registros de índice do que de dados, já reduz o número final de leituras realizadas. Além disso, dificilmente uma entrada de índice (que possui somente chave e ponteiro) tem tamanho próximo ao registro de dados (que contém todas as informações relevantes).

Terminada esta apresentação inicial sobre índices e consumo de espaço em disco, serão tratados na sequência duas formas de arquivos indexados: os com índice primário e os com índice secundário.

Índices primários

Um índice primário é um índice (que é um arquivo ordenado) para um arquivo de dados também ordenado. Diferentemente do exemplo dado acima, um índice primário não possui uma entrada de índice para cada registro do arquivo principal, mas sim um subconjunto das chaves.

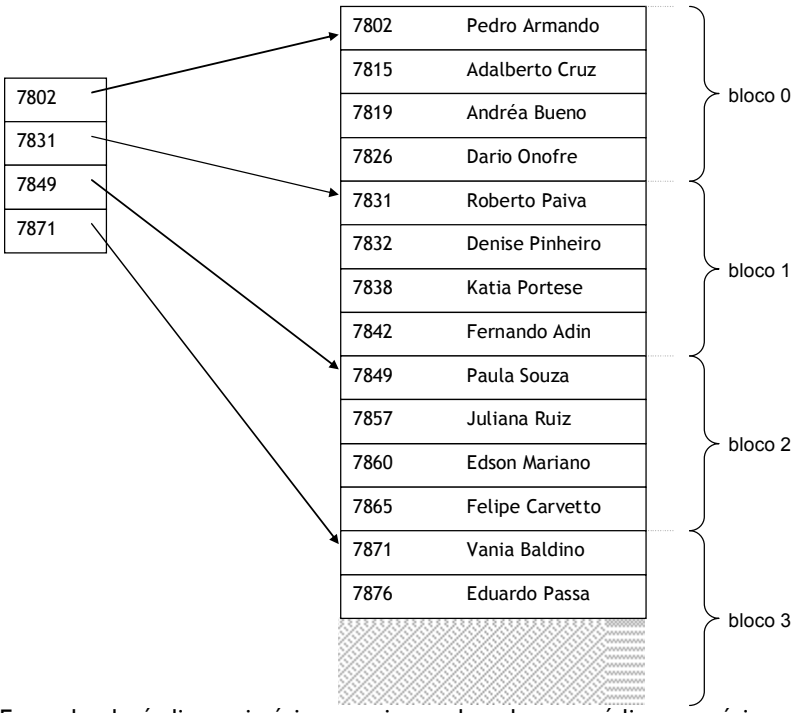


Figura 5-6. Exemplo de índice primário: arquivo ordenado por código numérico, contendo quatro registros inteiros por bloco, e indexado pelo mesmo campo numérico.

Índices primários são **índices esparsos**, ou seja, possuem número de entradas de índice menor que o número de registros de dados. Isso acontece devido a cada entrada de índice indicar um bloco, e não um registro. Ao conter apenas a primeira chave existente no bloco, o índice tem informações suficientes para localizar qualquer registro.

Por exemplo, a busca pela chave 7849 é iniciada por uma busca no índice, que tem seus blocos lidos para o buffer. Por ser um arquivo ordenado, a pesquisa binária pode ser empregada. A chave em questão é localizada no índice e o ponteiro com o endereço do bloco é utilizado para, em um único acesso a disco, recuperar os quatro registros do bloco. Finalmente o registro com chave 7849 é localizado entre os registros obtidos e a informação desejada é recuperada.

De forma similar, a chave 7860, para ser usada na localização, é procurada no índice. Como não há nenhuma entrada no índice com esta chave, não é difícil verificar que, se o registro com a chave existir, ele necessariamente estará no bloco 2, já que a próxima entrada de índice informa que todas as chaves no bloco subsequente são maiores ou iguais a 7871. O bloco apontado pelo índice é recuperado e, entre os quatro registros, a informação desejada é recuperada. Caso a chave buscada não existisse no arquivo de dados, então somente um bloco deste arquivo seria recuperado e analisado.

Um exemplo ajuda a compreender as vantagens desta estruturação. Para isso, retomam-se as características dos arquivos no exemplo da página 5-10. Supondo agora um arquivo de dados com 100.000 registros, seriam ocupados 12.500 blocos de disco. Como deve haver uma entrada de índice por bloco, serão necessários 12.500 registros no arquivo de índice. O arquivo de índice ocupará, então, 98 blocos (sendo o último não completo). Ao se fazer uma pesquisa binária, pode-se grosseiramente dizer que seriam necessárias $\log_2 98$, ou 6,61, acessos a disco para a busca no índice, embora esse número deva ser menor. Em outras palavras, com um máximo de 7 acessos a disco no índice e mais um acesso no arquivo de dados, qualquer um entre os 100.000 registros pode ser localizado. 100.000 registros com um máximo de 8 acessos a disco torna-se uma boa perspectiva de desempenho.

As vantagens da pesquisa binária e o uso do índice, porém, são penalizadas pelas demais operações. Inserções e remoções no arquivo de dados exigem movimentação média de metade dos registros existentes (no exemplo acima, 6.250 blocos reescritos por inserção ou remoção, em média). É importante lembrar, também, que a cada operação metade do arquivo de índice também tem que ser atualizado. Este é um custo alto e tem que ser considerado na escolha por esta forma de organização.

Índices secundários

Uma alternativa para os índices primários são os chamados **índices secundários**. Os índices secundários continuam sendo arquivos ordenados, para permitir uma pesquisa eficiente, mas tiram do arquivo de dados a restrição da ordenação. Assim, índices secundários são índices para arquivos não ordenados.

Uma das vantagens das tabelas em memória secundária serem mantidas não ordenadas é que as inserções (no final) e as remoções (cópia do último registro sobre o removido) são operações de baixo custo se comparadas ao caso dos arquivos ordenados. A desvantagem dos arquivos não ordenados é a pesquisa, que tem que ser sequencial. Ao se usar um índice para um arquivo não ordenado, o problema da pesquisa ineficiente é superado pela rápida pesquisa no índice. Deste modo, o arquivo de dados tem inserções e remoções eficientes, além de contar com uma pesquisa rápida usando o índice.

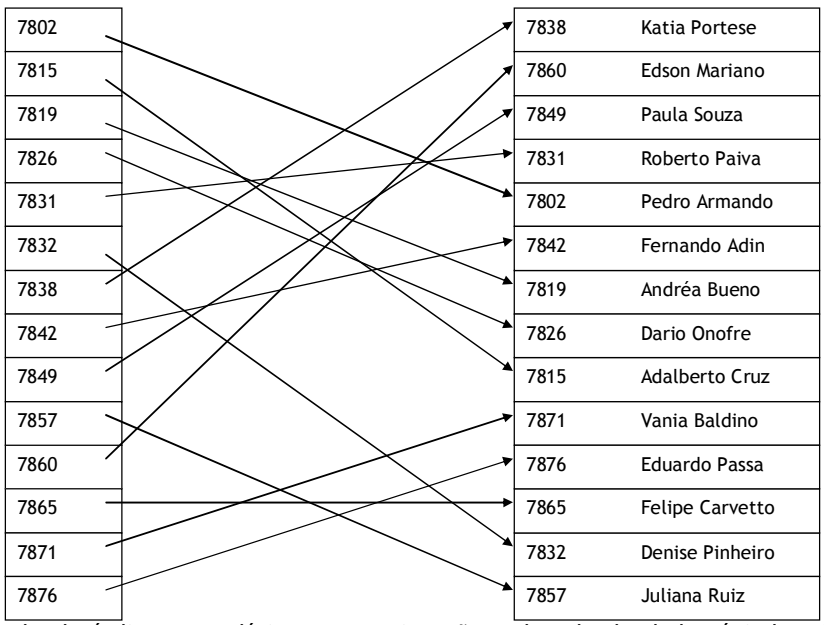


Figura 5-7. Exemplo de índice secundário: um arquivo não ordenado de dados é indexado por um arquivo ordenado de chaves e ponteiros para os registros individuais.

Porém, o arquivo do índice continua tendo que ser ordenado e sofre a penalização dos deslocamentos de registros quando houver inserções ou remoções de entradas. Outro ponto é que, sendo o arquivo de dados não ordenado, uma entrada de índice por bloco não é mais suficiente para localizar os registros. Deste modo, índices secundários têm que ser **densos**, ou seja, possuir uma entrada para cada registro de dados. A Figura 5-7 esquematiza um arquivo não ordenado com um índice denso.

Retomando os cálculos para um arquivo de 100.000 registros, nas mesmas condições dos exemplos anteriores, seriam necessários 782 blocos para armazenar as 100.000 entradas de índice. A manutenção do índice ordenado ainda demandaria a movimentação média de 391 blocos em média a cada inserção ou remoção. Por outro lado, o arquivo de dados ocuparia 12.500, mas cada inserção reescreveria apenas o último bloco e uma remoção afetaria apenas o bloco que contém o registro sendo removido e o último bloco do arquivo. No final de todas as contas, esta organização permite aliar as vantagens dos arquivos ordenados e dos não ordenados, tentando reduzir as desvantagens de ambos.

A principal restrição do uso de índices secundários é manter o índice permanentemente ordenado. Há, entretanto, uma proposição para contornar esta dificuldade. Esta solução gerencia uma estrutura que mantém as entradas de índice organizadas para permitir a pesquisa rápida, mas a ordenação não precisa ser física. Em outras palavras, o arquivo de índice é estruturado com ponteiros, os quais permitem realizar buscas de forma eficiente, mas os blocos apontados podem estar em qualquer parte do arquivo. A estrutura de dados em questão é uma árvore de vários caminhos conhecida como **árvore B**. Este texto não cobre árvores B, a qual é tratada em material separado.

5.4 - Considerações finais

As tabelas armazenadas em disco rígido possuem características semelhantes àquelas mantidas em memória principal. Os aspectos relativos às transferências de dados para o buffer interno, porém, devem ser considerados. Saber o que acontece “nos bastidores” é de suma importância para poder determinar a melhor solução para um determinado problema.

Arquivos não ordenados são práticos e, não possuindo tamanho considerável, podem ser uma boa solução quando há muitas inserções e remoções, sendo a pesquisa não crítica em termos de tempo. Pesquisas em arquivos ordenados, por outro lado, são altamente eficientes, por exemplo usando a pesquisa binária ou mesmo as pesquisas por estimativa. As operações de inserção e remoção, porém, ficam restritas, pois a ordenação física deve ser mantida.

Os arquivos indexados, sejam estruturados como primários ou secundários, acrescentam maior agilidade às pesquisas, mas o custo de manter a estrutura adicional e o espaço requerido para armazená-la devem ser considerados. Quando a velocidade de acesso é importante, índices são amplamente utilizados.

Os arquivos com índice secundário são uma alternativa bastante importante e, na prática, a forma mais comum de armazenamento em sistemas de bancos de dados. O arquivo de dados é mantido não ordenado, o que o torna prático para inserções e remoções. O índice (que é ordenado por uma dada chave) proporciona o acesso mais eficiente aos registros. Um ponto importante para esta organização é a possibilidade de que índices diferentes podem ser mantidos para um mesmo arquivo não ordenado. Isso significa que pesquisas podem ser feitas para várias chaves diferentes, bastando que para cada novo registro inserido, todos os índices sejam atualizados para suas respectivas chaves.