Estrutura de Arquivos
(MC326A)

Prof. Alexandre Xavier Falcão

Segundo semestre de 2007

1 Objetivos do Curso

Este curso abordará conceitos e métodos relacionados à manutenção de grandes arquivos de dados em disco, onde o enfoque principal é a indexação eficiente dos dados. Serão abordados dois paradigmas de busca indexada— busca exata e busca por similaridade— e as estruturas de arquivos correspondentes. Ao final, o aluno deverá ter adquirido o conhecimento básico necessário para disciplinas como banco de dados e recuperação de informações.

2 Ementa

- Introdução à Estrutura de Arquivos
- Estruturas de Arquivos em Disco
- Acesso a Arquivos em Disco
- Gerenciamento de Espaço Disponível em Arquivo
- Índices e Listas Invertidas
- Processamento Co-Seqüencial
- Arquivos de Índices em Disco (Árvores B e B^*)
- Acesso Seqüencial Indexado (Árvores B^+)
- Indexação por Estruturas Métricas
- Indexação por Espalhamento

3 Bibliografia

- M.J. Folk and B. Zoellick, File Structures, Addison-Wesley, 1992.
- P. Zezula, P. Ciaccia and F. Rabitti, "M-tree: A Dynamic Index for Similarity Queries in Multimedia Databases", Tech. Report 07, http://www.ced.tuc.gr/Research/Reports/HERMES/Reports.htm
- P. Ciaccia and M. Patella, "Performance of M-tree, an Access Method for Similarity Search in Metric Spaces", Tech. Report 13, http://www.ced.tuc.gr/Research/Reports/HERMES/Reports.htm
- P. Ciaccia and M. Patella, "Bulk Loading the M-tree", Tech. Report 28, http://www.ced.tuc.gr/Research/Reports/HERMES/Reports.htm
- N. Ziviani, Projeto de Algoritmos com Implementações em Pascal e C (2a. ed.). Thomson (2004). ISBN 85-221-0390-9.
- R. Baeza-Yates e B. Ribeiro-Neto, Modern Information Retrieval. Addison-Wesley (1999) ISBN 020139829X.
- M. Farley, Building Storage Networks. McGraw Hill (1999). ISBN 0072120509.
- N. Miller File Structures Using Pascal. Addison-Wesley (1987). ISBN 080537082X. (Out of Print).
- B. J. Salzberg, File Structures: An Analytic Approach. Prentice Hall (1988). 013314691X (Out of print).
- P. E. Livadas, File Structures: Theory and Practice. Prentice Hall (1990). ISBN 0133150941 (Out of Print).

4 Características físicas do disco

- Um disco rígido é composto por uma pilha de **pratos** que armazenam os dados em ambas as faces, exceto os pratos superior e inferior.
- Cada prato é composto de **trilhas**.
- Cada trilha é composta de **setores**.
- As trilhas que ficam uma acima da outra compõem um **cilindro**.

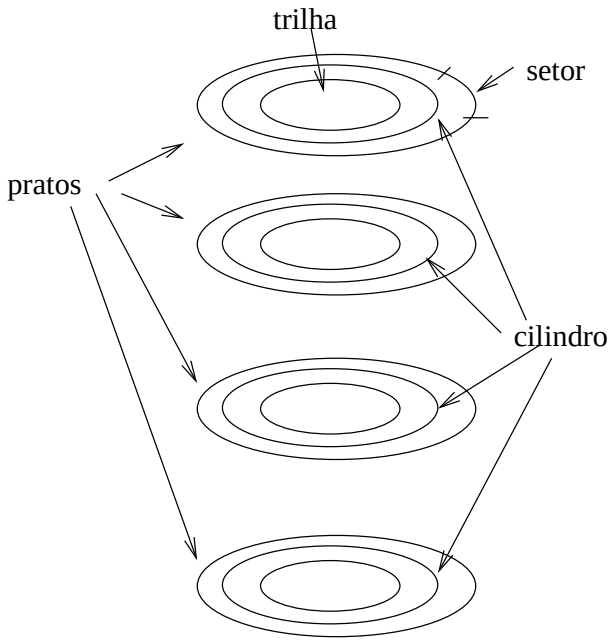

Figura 1: Características físicas de um disco rígido.

5 Armazenamento

Os arquivos são armazenados por cilindro, pois o tempo de mover o braço de leitura de um cilindro para outro é a parte mais cara do tempo de acesso aos dados (*seek time*).

O sistema operacional divide o disco em **blocos** de capacidade fixa de armazenamento. Neste caso, para que o número de blocos por trilha seja constante, as trilhas mais próximas do centro do prato são mais densas.

Exercício: Suponha que a capacidade de um bloco é 512 bytes, o número de blocos por trilha é 40, o número de trilhas por cilindro é 11, e o número de cilindros é 1000. Quantos cilindros são necessários para armazenar um arquivo com 10000 registros de 256 bytes cada?

1 cilindro = 11 trilhas x 40 blocos x 2 registros = 880 registros por cilindro. Então, o número de cilindros necessários é 10000/880 (aproximadamente 12 cilindros).

6 Leitura e gravação

Uma operação de leitura recupera vários blocos de uma só vez (e.g. uma **página** de 4Kbytes) e mantém esses blocos em uma área na memória principal. Alguns desses blocos são transferidos para memória *cache* e depois disponibilizados para o programa do usuário. O processo inverso ocorre no caso da gravação de arquivos contendo várias páginas. O usuário não tem controle sobre esses processos, visto que dependem do sistema operacional e do disco rígido da máquina. Porém, se souber o tamanho do bloco e o número de blocos por página, o usuário pode projetar o arquivo de forma a otimizar as operações de leitura e gravação (e.g. escolhendo o tamanho de

registro e o número de registros por nó da árvore B para que cada nó ocupe uma página). Isto é, bytes podem ser acrescentados aos registros para que seu tamanho seja tal que o número de registros por bloco seja inteiro.

O tempo de transferência A dos dados do disco para a memória principal depende do tempo de rotação B do disco, do número C de bytes por trilha, e do número D de bytes transferidos.

$$A = \frac{D}{C} \times B \tag{1}$$

7 Estruturas de arquivos

Quando escrevemos um programa para gerar um arquivo de dados (e.g. cadastro de pessoa física), costumamos agrupar os campos que fornecem informações sobre um mesmo indivíduo em um registro. Como o conceito de registro é lógico e não físico, a integridade dos campos e a integridade dos registros podem ser perdidas quando o arquivo é gravado em disco, impossibilitando a recuperação dos dados. Técnicas de organização dos campos e dos registros são usadas para manter a integridade deles em disco.

7.1 Organização dos campos

Considere um programa (**writestream.c**) para gravar um arquivo ASCII em disco com o primeiro nome e o último nome de cada indivíduo. Observe que a integridade de cada campo é perdida, pois os dados são armazenados como uma seqüência de bytes. Se tentarmos ler de volta os campos (**readstream.c**) e imprimí-los na tela, vamos observar que não conseguimos separá-los. Portanto, precisamos de alguma técnica para identificar os campos do arquivo.

- Forçar que cada campo ocupe um tamanho fixo em bytes.

  O tamanho do arquivo pode aumentar bastante caso algumas informações requeram campos compridos, mas na maioria dos casos os campos não sejam completamente preenchidos.

- Reservar um certo número de bytes antes de cada campo para indicar o seu comprimento.

  Se existirem campos com mais do que 256 bytes, por exemplo, a informação de comprimento requererá mais do que 1 byte. Se o arquivo for grande, este adicional de memória pode ser significativo.

- Inserir um delimitador separando os campos.

  Este delimitador pode ser um caracter (e.g. #), o qual ocupa apenas 1 byte, mas temos que cuidar para não selecionar um caracter comum aos campos

- Utilizar uma palavra chave para identificar o conteúdo de cada campo (e.g. nome=Alexandre).

  Este método tem a vantagem de prover informações sobre o conteúdo do arquivo, o que os outros métodos não oferecem, mas desperdiça bastante memória em disco.

Ver programas **writefields.c** e **readfields.c**.

7.2 Organização dos registros

- Registros de tamanhos fixos
  - Campos de tamanhos fixos

    Não precisamos nos preocupar com os delimitadores, pois as funções *fwrite* e *fread* vão gravar e ler o mesmo número de bytes definido para cada campo. Porém, esta opção desperdiça muita memória nos campos do registro (**fragmentação**).
  - Campos de tamanhos variáveis

    O número de campos pode ser variável no registro com delimitadores entre campos, não teremos desperdício no campo, mas podemos desperdiçar bytes do final do registro (fragmentação).

    Uma alternativa é definir o tamanho do registro supondo que os tamanhos máximos possíveis dos campos não ocorrem simultaneamente para um mesmo registro.
- Registros de tamanhos variáveis

  Os campos podem ser de tamanhos variáveis separados por delimitadores, porém precisamos identificar os registros também.
  - Reservar um certo número de bytes antes de cada registro para indicar o seu comprimento.
  - Inserir um delimitador separando os registros.
  - Usar um outro arquivo com o endereço em disco (*offset*) de cada registro.

A escolha do método depende da natureza e da utilização dos dados. Por exemplo, vamos supor o método de registros de tamanhos variáveis com campos de tamanhos variáveis, que indica o comprimemto de cada registro e separa os campos por delimitadores (ver programas **writerec.c** e **readrec.c**). Dois problemas neste método são: saber o comprimento do registro antes de gravá-lo em disco e saber o comprimento máximo dos registros para decidir se usamos um inteiro (4 bytes) ou uma string de n caracteres (n bytes) para gravar esta informação.

Para ler o registro, evitando acesso byte a byte ao arquivo, os bytes que correspondem ao registro podem ser lidos de uma só vez e carregados em um *buffer* na memória principal. O *buffer* é então interpretado byte a byte. O processo inverso pode ser feito para a gravação.

Para visualizar o conteúdo do arquivo em disco use $od\_ - xc\_ < nomedoarquivo >$.

8 Acesso aos dados

Cada registro pode ser unicamente identificado por uma "chave de acesso" (i.e. **chave primária**), que pode ser um número ou uma seqüência de caracteres.

Caso a unicidade não seja exigida, um grupo de registros poderá ser acessado com uma mesma chave, denominada **secundária**.

No caso de uma seqüência de caracteres, chaves primárias e secondárias devem ser representadas em forma canônica (padrão).
Ex: As chaves falcão, Falcão, FALCÃO terão representação única FALCAO para indicar um mesmo registro ou um grupo de registros.

Partimos do princípio que o arquivo de dados é muito grande e não cabe na memória principal. Seus registros devem ser acessados no disco.

8.1 Acesso seqüencial

A forma mais simples de acesso é seqüencial, i.e. o arquivo é lido registro por registro até encontrarmos o(s) registro(s) que possue(m) a mesma chave de acesso.
Ex: Quando usamos o comando grep no UNIX/LINUX.

Normalmente esta é a forma menos eficiente de acesso, porém algumas situações favorecem o acesso seqüencial.
Ex: Quando o arquivo possui poucos registros, quando se trata de uma chave secundária que recupera um alto número de registros, e quando o acesso é uma operação rara (e.g. atualização de um dado registro em um arquivo de backup).

8.2 Acesso direto

Visto que o acesso ao disco é computacionalmente bem mais caro do que o acesso à memória principal, o acesso seqüencial passa a ser um problema cuja solução requer acesso direto.

O acesso direto a um dado registro (ou grupo de registros) em disco requer conhecer o(s) endereço(s) do(s) registro(s) no arquivo de dados. Assim, comandos como fseek, podem ser usados para localizar diretamente o(s) registro(s) para leitura/gravação.

Esses endereços e suas respectivas chaves são armazenados em uma estrutura de dados, denominada **índice**. Dependendo do seu tamanho, o índice pode caber ou não na memória principal. O último caso requer que o índice seja armazenado em arquivo separado no disco e carregado por partes na memória principal, durante a busca.

8.3 Endereçamento

O endereço de um registro é o deslocamento em bytes (*offset*) desde o início do arquivo, ou final do **cabeçário do arquivo** (registro que armazena informações gerais tais como comprimento dos registros, data da última atualização, número de registros, etc.).

No caso de registros de tamanho fixo, podemos armazenar no índice apenas o número de registros que antecedem cada registro (**Relative Record Number - RRN**). Neste caso, o endereço de um dado registro é obtido multiplicando-se o seu RRN pelo comprimento dos registros.

9 Gerenciamento de espaço disponível em arquivo

Suponha que um registro de tamanho variável é modificado de tal forma que o novo registro fica mais longo do que o original. Como resolver o problema?

1. Colocar os dados extras no final do arquivo e usar um campo no registro para indicar o endereço desses dados.

2. Gravar todo o registro no final do arquivo e disponibilizar o espaço original para um novo registro menor.

A opção 1 demanda muito processamento. A opção 2 é mais atraente, mas requer a solução de dois novos problemas.

1. Como reconhecer que um certo espaço no arquivo está disponível?

   Podemos colocar uma marca (*) no primeiro campo do registro cujo tamanho indica agora o número de bytes disponíveis.

2. Como reutilizar o espaço disponível?

   (a) Solução estática: Podemos marcar os registros disponíveis e rodar de tempos em tempos um programa para copiar os registros ativos para um novo arquivo.

   (b) Solução dinâmica: Podemos marcar os registros disponíveis e criar uma lista ligada com seus endereços.

   Esta solução é mais atraente, pois pode usar o próprio espaço disponível no arquivo para armazenar a lista, provendo formas imediatas de saber se existe um espaço disponível (i.e. lista não vazia) e de acessar o endereço disponível. O nó cabeça pode ser armazenado no cabeçário do arquivo e os demais nos espaços disponíveis.

O gerenciamento de espaços disponíveis também ocorre no caso de remoção de registros, variando apenas o tratamento para registros de tamanho fixo e de tamanho variável.

9.1 Gerenciamento de memória com registros de tamanho fixo

A lista deve ser uma pilha, onde cada nó tem a marca e o RRN do próximo registro disponível. Usa-se sempre o primeiro disponível na pilha (Figura 2a).

9.2 Gerenciamento de memória com registros de tamanho variável

Cada nó tem a marca, o endereço do próximo espaço disponível e o tamanho do espaço corrente (Figura 2b). Esta abordagem traz dois novos problemas.

1. O espaço disponível tem que ter tamanho mínimo necessário para acomodar o novo registro (i.e. não podemos usar uma pilha e devemos buscar na lista o espaço a ser usado).

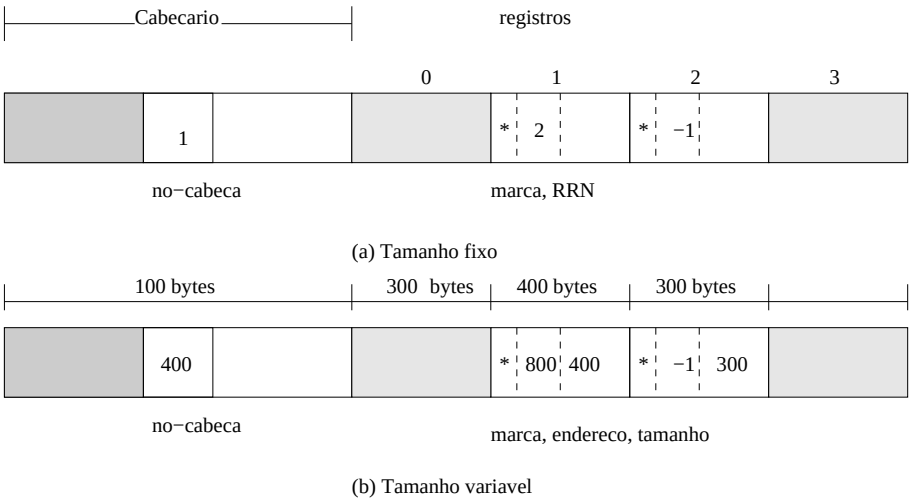

Figura 2: Listas de endereços disponíveis para arquivos com registros de tamanho (a) fixo e (b) variável.

2. Se o novo registro for menor que o espaço disponível para ele, teremos fragmentação no final deste espaço. Caso o espaço que sobra seja colocado de volta na lista, ele pode ser pequeno demais para ser reutilizado no futuro (i.e. a fragmentação persiste). Neste caso, podemos ainda fazer a união de espaços disponíveis adjacentes, mas isto requer processamento adicional, na inserção, para identificar esses espaços.

Esses problemas levam as seguintes técnicas de gerenciamento de memória.

1. First-fit: Varre-se a lista (pilha) e o primeiro espaço disponível com tamanho suficiente é usado. Não requer processamento adicional e é a opção mais indicada no caso de registros com mais ou menos o mesmo tamanho. A remoção insere o registro no início da lista.

2. Worst-fit: Mantém-se os nós da lista em ordem decrescente de tamanho. A vantagem é evitar varrer o resto da lista quando o primeiro nó já não tem tamanho suficiente para acomodar o novo registro. É a melhor opção se os espaços disponíveis são pequenos. Porém, desperdiça mais espaço no registro do que seria necessário e requer processamento adicional.

3. Best-fit: Mantém-se os nós da lista em ordem crescente de tamanho. Minimiza o desperdício de espaço no registro e é uma boa opção no caso de registros com mais ou menos o mesmo tamanho. Porém, requer processamento adicional.

10 Índices e listas invertidas

Um **índice** é uma estrutura de dados contendo dois campos: uma **chave** e uma **referência**. A chave está associada a um campo (ou combinação de campos) do registro de um arquivo

de dados. Um índice é dito **primário**, quando a chave identifica um único registro apenas no arquivo de dados, e é dito **secundário** no caso de múltiplos registros. Em ambos os casos, devemos adotar a forma canônica para a chave. A referência em índice primário indica o endereço do registro cujo conteúdo do campo é igual ao conteúdo da chave. No caso de índice secundário, a referência é a chave primária.

10.1 Índice primário

A referência pode ser em RRN, para arquivos com registros de tamanho fixo, e em bytes para arquivos com registros de tamanho variável. Inicialmente, vamos adotar uma tabela como estrutura de indexação. A Figura 3 ilustra um índice primário.

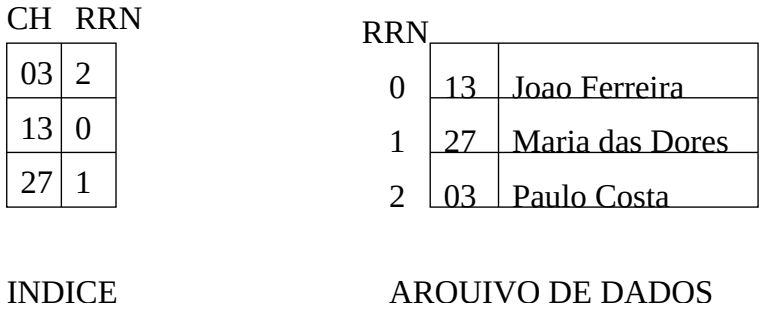

Figura 3: Índice primário.

O índice e os dados são mantidos em arquivos separados. Uma vantagem é que o índice é bem menor e pode ser mantido ordenado para acelerar a busca. O índice deve possuir registros com tamanho fixo, e se for pequeno o suficiente, pode ser mantido em memória primária.

Note, porém, que no caso de uma falta de luz ou qualquer outra interrupção na execução do programa, o índice em memória primária pode ser perdido. Neste caso, o índice em disco ficará desatualizado com relação ao arquivo de dados e precisará ser refeito. O índice em disco deve ser atualizado de vez em quando, por segurança, e uma marca indicando desatualização pode ser apagada do seu cabeçário toda vez que for atualizado. Assim saberemos se o índice está ou não desatualizado, no caso de interrupção do programa.

No caso da manutenção do índice ser em disco, estruturas mais eficientes, tais como a árvore B, são necessárias.

10.2 Índices secundários

Um arquivo de dados pode ser indexado por várias chaves secundárias. A Figura 4 ilustra um exemplo com dois índices secundários. Esses arquivos formam listas invertidas, onde o índice primário é usado para acessar os registros no arquivo de dados.

Uma busca por JOAO encontra dois registros no índice secundário de nome. Cada registro gera uma nova busca no índice primário e por fim um acesso ao registro no arquivo de dados, formando uma lista invertida. O resultado fica:

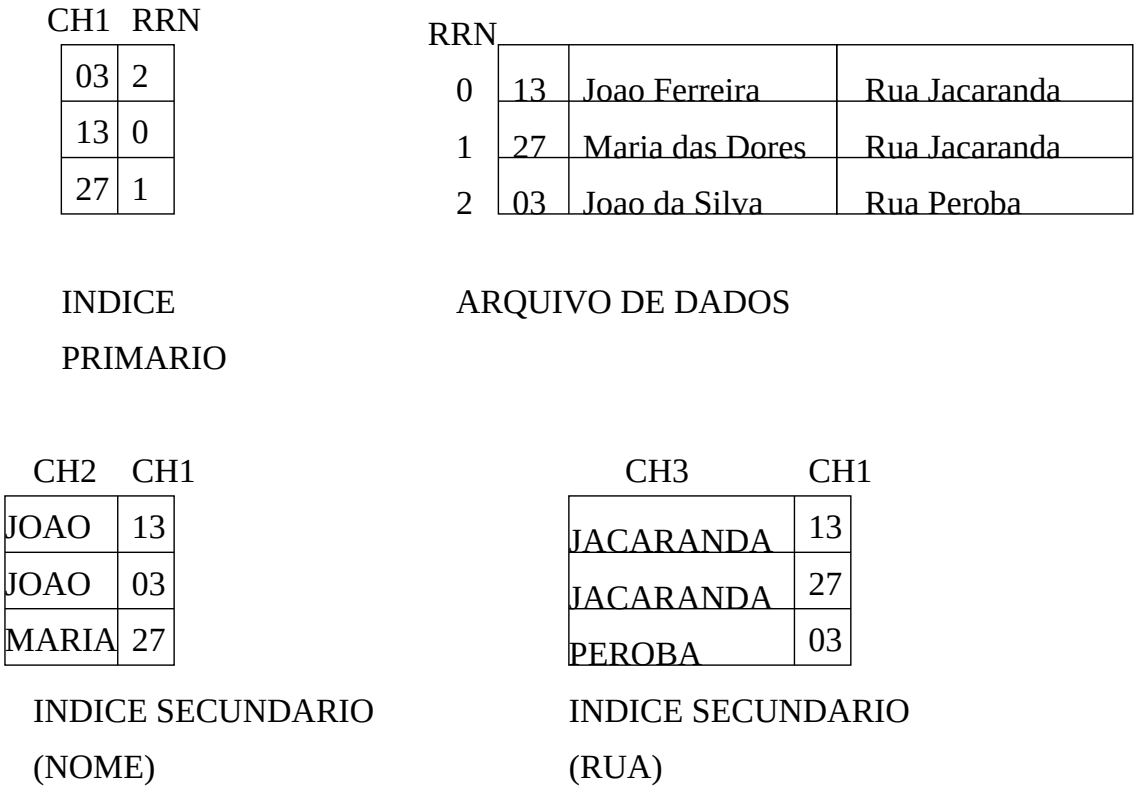

Figura 4: Índices secundários formando listas invertidas.

13, João Ferreira, Rua Jacarandá
03, João da Silva, Rua Peroba

Operadores OR e AND podem ser usados para refinar a busca. A busca por JOAO AND JACARANDA, por exemplo, só resulta em um registro.

13, João Ferreira, Rua Jacarandá

Observe que, a remoção de um registro no arquivo de dados implicaria na atualização de todos os índices. Isto pode ser evitado com uma marca no índice primário, deixando o registro do índice primário na lista de disponíveis. Um acesso pelo índice secundário pára quando encontra a marca de removido. Porém, os índices secundários devem ser atualizados assim que o registro do índice primário for usado para armazenar outro par chave-referência.

Podemos evitar também que o índice secundário tenha que ser rearranjado toda vez que um registro com mesma chave secundária for acrescentado ao arquivo de dados, evitar a repetição da chave secundária, e ao mesmo tempo, não impor limite ao número de chaves primárias associadas a uma chave secundária. A solução separa o índice secundário em dois arquivos com registros de tamanho fixo, conforme ilustra a Figura 5.

Figura 5: Índices secundários formando listas invertidas.

11 Processamento co-seqüencial

É o processamento coordenado de duas ou mais listas em disco (e.g. tabelas de índices) para produzir uma única lista em disco. As operações básicas são matching (intersecção), merging (união), e sorting (ordenação) de listas.

11.1 Intersecção

Suponha que desejamos gerar uma terceira lista de nomes com os nomes em comum em duas listas de entrada, assumindo que não existem nomes duplicados nas listas de entrada e que estas estão ordenadas em ordem crescente de nome (e.g. poderiam ser tabelas de índices primários).

```
1. n1 = read_name(list1);
2. n2 = read_name(list2);
3. while (!eof(list1) && !eof(list2)) {
4.   if (n1 < n2)
5.     n1 = read_name(list1);
6.   else
7.     if (n1 > n2)
8.       n2 = read_name(list2);
9.     else{ /* n1 == n2 */
10.      write_name(n1,list3);
11.      n1 = read_name(list1);
```

```
12.      n2 = read_name(list2);
13.    }
14. }
```

Observe que o sincronismo é mantido e que o algoritmo termina ao final da menor lista.

11.2 União

No caso da união de duas listas ordenadas, devemos observar que um nome duplicado deve ser gravado na lista de saída uma única vez e que ambas as listas devem ser lidas até o final.

```
1. n1 = read_name(list1);
2. n2 = read_name(list2);
3. while (!eof(list1) || !eof(list2)) {
4.   if (n1 < n2) {
5.     write_name(n1,list3); n1 = read_name(list1);
6.   }else
7.     if (n1 > n2){
8.       write_name(n2,list3); n2 = read_name(list2);
9.     }else{ /* n1 == n2 */
10.      write_name(n1,list3);
11.      n1 = read_name(list1);
12.      n2 = read_name(list2);
13.    }
14. }
```

O algoritmo acima retorna o maior nome possível, quando a lista já está finalizada. Este nome nunca ocorre nas listas de entrada. Genericamente, podemos ainda unir $k > 2$ listas.

```
1. for (i=0; i < k; i++)
2.   n[i] = read_name(list[i]);
3. continua = 0;
4. for (i=0; i < k; i++)
5.   continua = continua || (!eof(list[i]));
6. while (continua) {
7.   out_n = MIN(n); /* menor nome no vetor n */
8.   write_name(out_n,out_list);
9.   for (i=0; i < k; i++) {
10.    if (n[i]==out_n)  /* pode existir mais de um mínimo */
11.  n[i]=read_name(list[i]);
```

```
12.  }
13.  continua = 0;
14.  for (i=0; i < k; i++)
15.    continua = continua || (!eof(list[i]));
16. }
```

Se o número de listas passar de 8, podemos otimizar a função MIN(n) usando uma árvore de seleção em memória principal (ver Figura 6). A função passa a ter complexidade $O(\log_2^k)$ para manter a árvore de seleção.

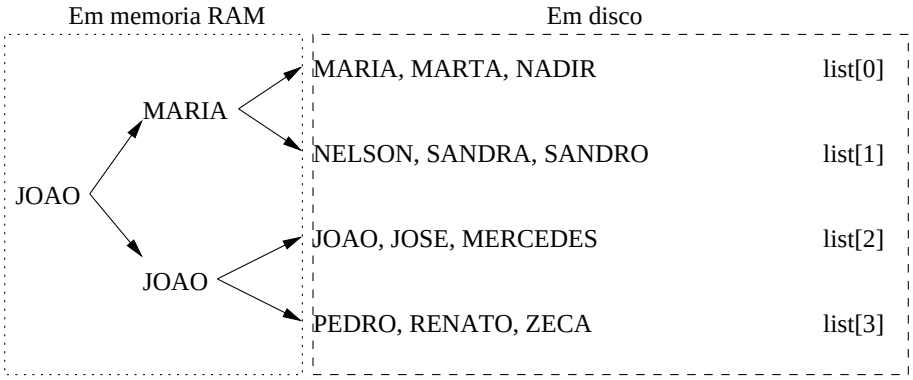

Figura 6: Árvore de seleção.

11.3 Ordenação

Podemos usar a união de k listas para ordenar arquivos em disco. O arquivo deve ser dividido em k partes que caibam individualmente na memória principal. Cada parte é carregada na memória principal, ordenada, e depois gravada em um arquivo separado em disco. O algoritmo de união é então usado para unir os k arquivos mantendo a ordem entre eles.

Uma alternativa preliminar seria o algoritmo *keysort*. Este algoritmo carrega na memória principal registros com as chaves primárias e suas respectivas referências (endereços do registro no arquivo); ordena esses registros pelas chaves; e depois procura no disco os registros correspondentes, seguindo a ordem das chaves, para gravá-los em um novo arquivo ordenado. Porém, esta idéia apresenta as seguintes desvantagens:

- Para cada chave temos que procurar o respectivo registro no disco.
- Pode ser que todas as chaves e referências do arquivo não caibam na memória principal.

Portanto, a alternativa que usa a união de k listas é melhor. A questão é qual algoritmo devemos usar para ordenação das k partes em memória principal?

O *heapsort* oferece a melhor solução. Pois não requer que todos os registros sejam lidos para depois ordená-los (a ordenação ocorre em paralelo com a leitura, uma vez que cada registro lido é colocado na ordem correta no *heap*. Enquanto o *heap* é ajustado, o programa está lendo o próximo registro), não requer memória extra (ordenação *in-place*), e não requer alocação dinâmica. O mesmo vale para a gravação, onde os registros são retirados na ordem correta, e enquanto são gravados em disco, o *heap* pode ser ajustado para a retirada do próximo registro. Você pode programar esse algoritmo usando multi-threads.
Exercício:

Suponha as informações abaixo e responda: qual é o tempo esperado para ordenar o arquivo em disco? Despreze o tempo de processamento em memória principal.

- Seja um arquivo com 20.000.000 registros de 1.000 bytes cada;
- Uma memória principal com 1.000.000.000 bytes, dos quais a metade pode ser usada para buffering na etapa de união de listas;
- O tempo médio de busca em disco, incluindo o tempo de rotação da cabeça de leitura, é de 10ms.
- O sistema transfere blocos de 100 registros a uma taxa de 10.000 bytes por ms.

Podemos carregar na memória principal $\frac{1.000.000.000}{1.000} = 1.000.000$ de registros por vez, ordená-los, e gravá-los de volta no disco. Portanto, o arquivo pode ser dividido em $k = \frac{20.000.000}{1.000.000} = 20$ partes. O tempo de transferência para cada bloco de 100 registros é $\frac{100*1.000}{10.000} = 10$ms. Considerando que o tempo médio de busca desses registros é 10ms, então o tempo gasto para carregar cada 100 registros na memória principal será 20ms. Cada parte necessitará de $\frac{1.000.000}{100} = 10.000$ operações de leitura, portanto, o tempo total de leitura será $10.000 * 20ms = 200$s. A gravação em disco terá o mesmo custo de 200s. O tempo gasto para ler, ordenar em memória, e gravar cada parte será $20 * 400 = 8.000$ segundos (cerca de 2,2 horas).

Para unir os 20 arquivos ordenados, podemos dispor de 500.000.000 bytes de memória. Isto nos dá um buffer de 25.000.000 de bytes por arquivo. Como cada registro tem 1.000 bytes, então podemos carregar na memória $\frac{25.000.000}{1.000} = 25.000$ registros de cada arquivo por vez. Cada 100 registros são lidos em 20ms, então 25.000 registros serão lidos em $\frac{25.000*20}{100} = 5$ segundos. Como cada arquivo ordenado possui 1.000.000 de registros, o tempo gasto com a leitura de cada arquivo será de $\frac{1.000.000*5}{25.000} = 200$ segundos. Como são 20 arquivos, o tempo total de leitura será 4000 segundos. À medida que unimos os arquivos, os registros ordenados podem ser armazenados na outra metade da memória. Este buffer de saída tem capacidade para armazenar $\frac{500.000.000}{1.000} = 500.000$ registros. Isto significa que faremos $\frac{20.000.000}{500.000} = 40$ operações de gravação de 500.000 registros cada. Como cada 100 registros são gravados em 20ms, a gravação de 500.000 registros levará $\frac{500.000*20}{100} = 100$ segundos. O tempo total para gravação será portanto de 4000 segundos.

O tempo total de ordenação será, portanto, de $8.000 + 4.000 + 4.000 = 16.000$ segundos (cerca de 4.44 horas). Repita esta análise considerando agora 75% do buffer para carregar as listas ordenadas e 25% do buffer para a lista de saída.

12 Árvores B (R. Bayer e E. McCreight, 1972)

12.1 Introdução

Inicialmente, os arquivos de índices eram mantidos em disco como tabelas ordenadas. A busca binária nessas tabelas era ineficiente, pois envolvia o acesso a várias trilhas. Manter a tabela ordenada também nem sempre era eficiente. A primeira solução foi usar árvores binárias de busca, em seguida árvores AVL, e depois árvores B.

O tempo de busca foi consideravelmente reduzido com essas árvores, acomodando vários nós (registros) em uma página de memória. Por exemplo, imagine uma árvore binária com 7 nós por página (Figura 7). São necessários 2 acessos a disco para localizar qualquer um de 63 nós, 3 acessos para 511 nós, 12 acessos para 4095 nós, e assim por diante. Em relação às tabelas, o número de acessos caiu de $\log_2^{(N+1)}$, onde N é o número total de registros, para $\log_{(K+1)}^{(N+1)}$, onde K é o número de registros por página. Isto significa que para uma árvore com 134217727 registros e 511 registros por página, qualquer registro pode ser encontrado com 3 acessos a disco (redução de 27 para 3).

O principal problema das árvores AVL, que motivou o desenvolvimento das árvores B, decorre do algoritmo de construção da árvore ser de cima para baixo. Durante a construção de uma árvore AVL, os registros agrupados em uma mesma página e seqüencialmente gravados em disco são constantemente rotacionados e/ou transferidos para novas páginas, para manter o balanceamento. Na árvore B, além do problema ser menor com a construção de baixo para cima, cada nó da árvore corresponde a uma página.

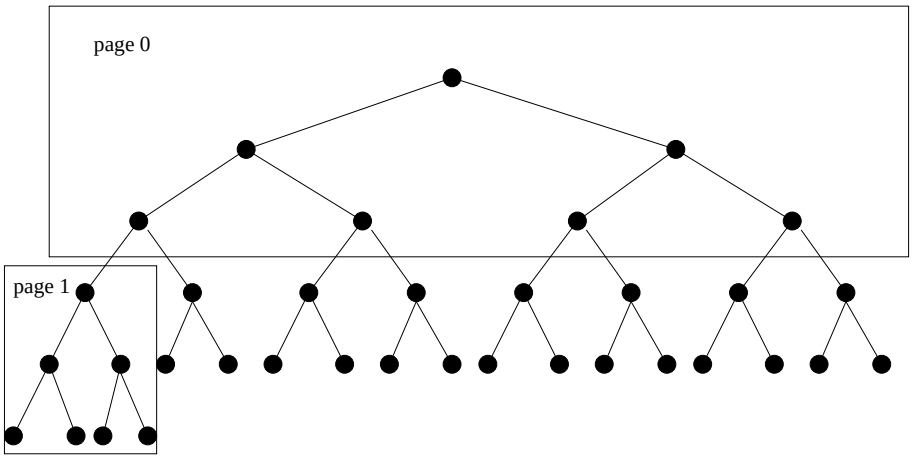

Figura 7: Paginação em árvore binária.

12.2 Implementação da árvore B em disco

Vimos no curso anterior como implementar uma árvore B em memória. Neste curso, os índices podem ser representados por árvores B, que supostamente não cabem na memória, e portanto,

devem ser mantidas como um arquivo binário em disco. Vamos considerar inicialmente o caso de **índice primário**.

- Cada nó da árvore (página em memória ou bloco de registros no disco) pode ser dimensionado para ser lido/gravado com um único acesso a disco. Observe que os registros são de **tamanho fixo**, e conseqüentemente, os blocos são de tamanho fixo. Portanto, o arquivo índice está dividido em **blocos de tamanho fixo**, os quais possuem endereço dado em RBN (*Relative Block Number*).

- O endereço do bloco raiz é armazenado no cabeçário do arquivo.

- Um bloco contém um número K máximo de registros, e não menos que $\max\{K/2, 1\}$ registros, exceto a raiz que pode ter no mínimo 1 registro. Os campos de um nó são o número de registros armazenados, os registros ordenados pela chave primária, e os endereços (em RBN) dos blocos descendentes.

- Cada registro armazena uma chave primária e o endereço do registro correspondente no arquivo de dados, que pode ser de tamanho fixo ou variável.

- A busca por um registro do arquivo de dados requer carregar em memória o nó raiz e buscar dentro do nó o registro que contém a chave primária de busca. Se o registro for encontrado, então os dados correspondentes são acessados diretamente no arquivo de dados. No caso contrário, o nó descendente, onde o registro pode ser encontrado, é carregado em memória e a busca se repete de forma recursiva.

- A inserção e a remoção também carregam as páginas na memória à medida que vão precisando delas. Na construção da árvore, por exemplo, a primeira página contendo um único registro é gravada em disco. À medida que surgem novas inserções, a página é recarregada na memória, o registro é inserido, e a página é gravada de volta no disco. Quando a inserção não pode ser acomodada na página, a página se divide em duas, o registro com chave mediana é inserido em uma terceira página, e as três são gravadas em disco (Figura 8).

- A divisão e a união de páginas causadas por inserção e remoção de registros fazem com que mais de uma página precise ser carregada em memória ao mesmo tempo. Note que isto ocorre na volta da recursão, onde sabemos o endereço da página pai e das páginas descendentes.

- A remoção de um bloco pode ser indicada com uma marca e o bloco deve ser inserido em uma lista de blocos disponíveis, cujo endereço do início da lista está armazenado no cabeçário do índice primário.

- Novas inserções devem aproveitar os blocos disponíveis, sendo feitas no final do arquivo, apenas quando necessário.

A Figura 9 ilustra mais alguns casos de inserção e remoção na árvore B da Figura 8.

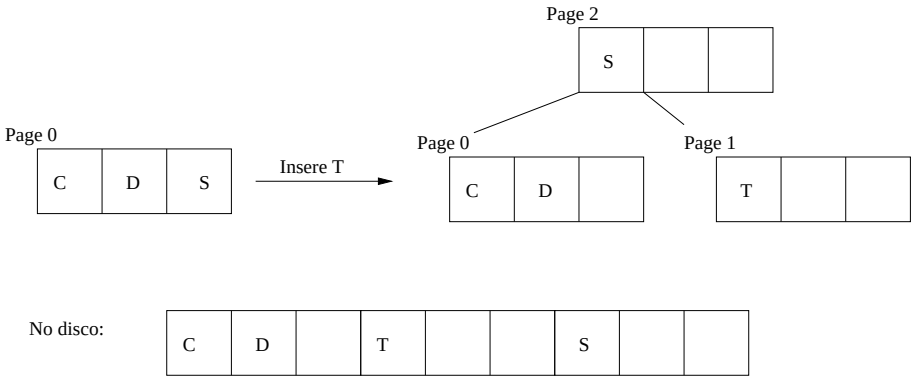

Figura 8: Inserção com divisão em árvore B, onde as chaves primárias são letras e $K = 3$.

A manutenção em disco de um **índice secundário** é muito parecida com a de um índice primário, exceto por situações decorrentes do fato que o índice secundário é dividido em dois arquivos: um primeiro arquivo (árvore B), dividido em blocos de tamanho fixo, e um segundo arquivo (listas ligadas) dividido em registros de tamanho fixo, o qual evita repetição de chaves secundárias no primeiro.

- Cada registro do primeiro arquivo possue uma **chave secundária** e o **endereço** (em RRN) do primeiro registro com esta chave no segundo arquivo.
- Cada registro do segundo arquivo possue uma **chave primária** e o endereço (em RRN) do próximo registro neste arquivo com a mesma chave secundária.
- Ambos arquivos devem possuir uma lista de blocos e de registros disponíveis, respectivamente.

Outros aspectos importantes na manutenção desses índices em disco são:

- Manter uma distribuição uniforme do número de registros por página em uma árvore B pode ser uma estratégia para reduzir o número de divisões e uniões, e conseqüentemente, o número de acessos a disco. Este processo requer uma redistribuição de registros entre páginas irmãs durante uma inserção ou remoção, mesmo que não haja necessidade de divisão ou união.

  Suponha, por exemplo, uma remoção em uma página com 50 registros, quando a página irmã possui 100 registros, e a ordem da árvore é 101. Em vez de pegar um único registro emprestado da irmã, nós pegamos 25 registros.

  Um variante, denominado árvore B*, utiliza esta estratégia durante as inserções para manter um mínimo de 2/3 da capacidade da página em número de registros. Quando há necessidade de uma divisão, normalmente a página irmã também está cheia, e portanto, ela pode emprestar registros para manter o requisito de 2/3 da capacidade por página (ver Figura 10).

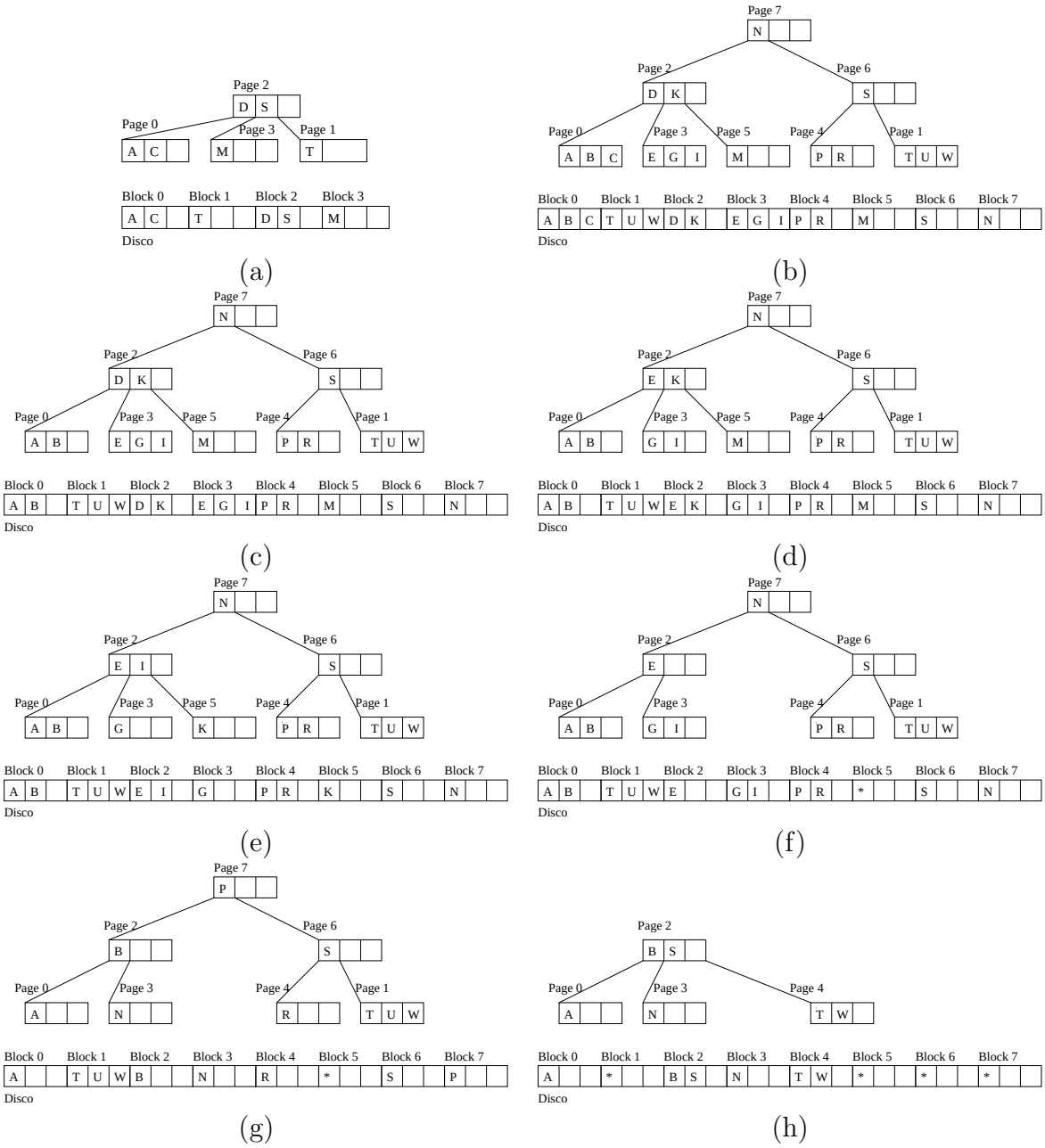

Figura 9: Inserção de (a) A,M, e depois de (b) P,I,B,W,N,G,U,R,K,E. Remoção de (c) C é resolvida na própria página, de (d) D requer troca com chave sucessora em nó folha antes da remoção, de (e) M requer empréstimo do irmão mais rico, de (f) K requer união de nós, de (g) G,I,E propaga o underflow mas não reduz a altura da árvore, e de (h) P,U,R reduz a altura da árvore.

- Outro variante é a virtual B tree que carrega várias páginas de uma só vez na memória para minimizar o número de acessos a disco.

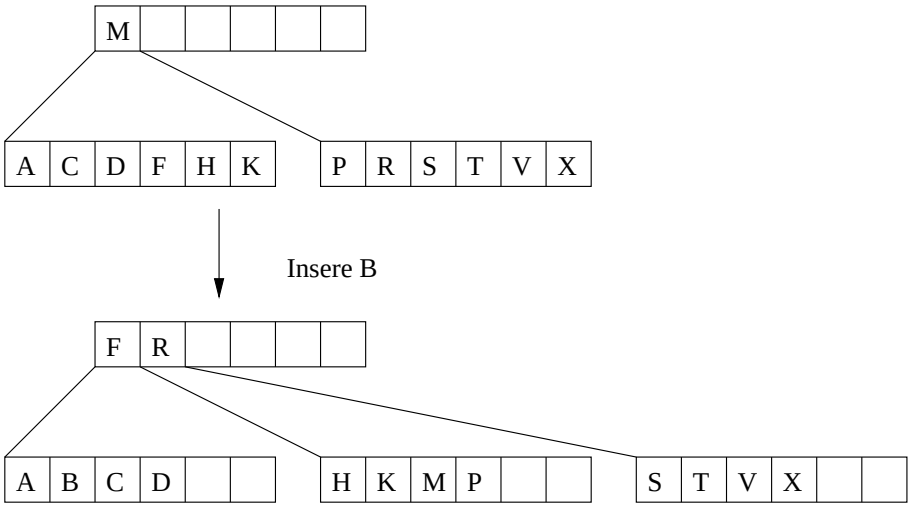

Figura 10: Inserção com divisão e redistribuição em árvore B*.

13 Acesso seqüencial Indexado: Árvores B$^+$

13.1 Motivação

Muitas aplicações requerem acesso seqüencial ordenado a todos os registros do arquivo para atualizar ou imprimir informações (e.g. união de índices), mas também requerem acesso individual para consulta, inserção ou remoção de um dado registro.

13.2 Árvore B$^+$

- Os registros do arquivo são inseridos em ordem crescente de chave e agrupados por blocos em disco (páginas em memória), formando um **conjunto de seqüências de registros** (Figura 11). Por exemplo, todas as chaves do bloco i são menores que as do bloco $i+1$ para $i = 0, 1, \ldots, n-1$.
- Os blocos do conjunto são duplamente ligados usando os endereços em bytes do próximo e do anterior.
- Algumas idéias de árvores B são incorporadas para otimizar o uso de memória.
  - Cada bloco pode ter um número mínimo e um número máximo de registros.
  - Uma inserção pode gerar divisão, com criação de um novo bloco, e redistribuição de registros (Figura 12).
  - Uma remoção pode gerar uma concatenação de blocos irmãos com disponibilização de um dos blocos para futuras inserções (Figura 13).

  - A redistribuição de registros pode ser feita sempre que um bloco tenha mais que metade da sua capacidade e exista um vizinho com menos que metade da sua capacidade.

- O acesso a um dado bloco é direto através de uma **árvore B**, onde cada nó (página em memória) possui um vetor de **separadores** de blocos e um vetor endereços dos nós descendentes. Os separadores são as chaves dos primeiros registros de cada bloco. O tamanho de cada nó deve ser igual ao tamanho de cada bloco do conjunto de seqüências. Assim, os endereços nos nós internos são dados em RBN (*relative block number*), sendo que os nós folha terão como descendentes os blocos do conjunto (Figura 14). O acesso a um registro, portanto, requer caminhar na árvore B até um nó folha e depois carregar em memória o bloco correspondente.

- As inserções, remoções e distribuições no conjunto de seqüências podem ou não alterar os separadores. As alterações nos separadores são realizadas usando as rotinas de manutenção de árvores B.

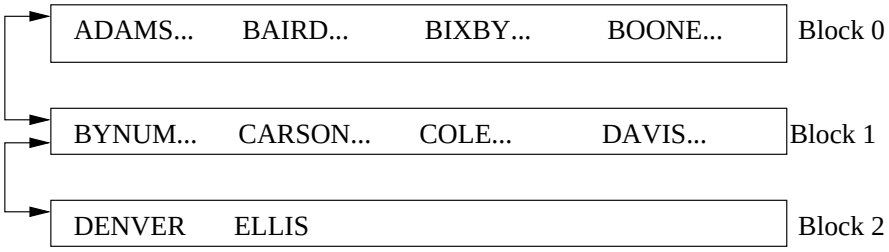

Figura 11: Conjunto de seqüências de registros.

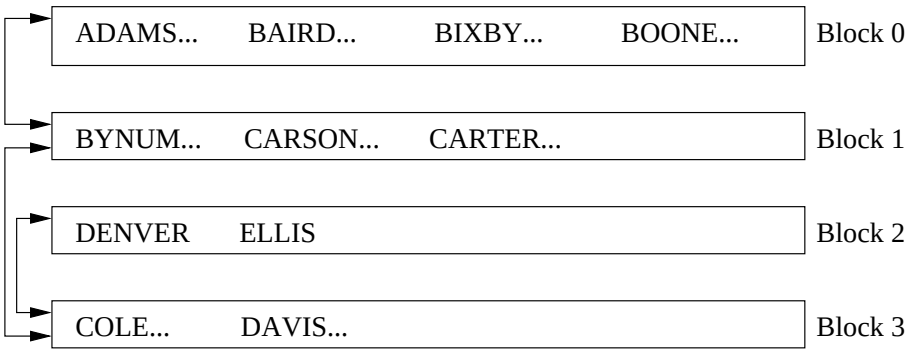

Figura 12: Inserção de CARTER com divisão no conjunto de seqüências de registros da Figura 11.

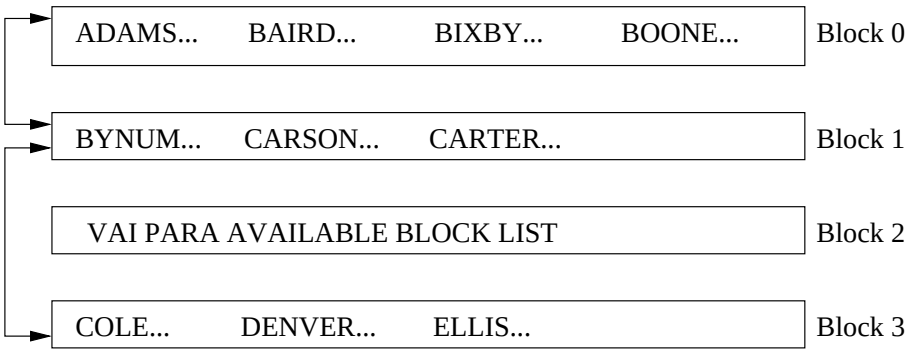

Figura 13: Remoção de DAVIS com concatenação no conjunto de seqüências de registros da Figura 12.

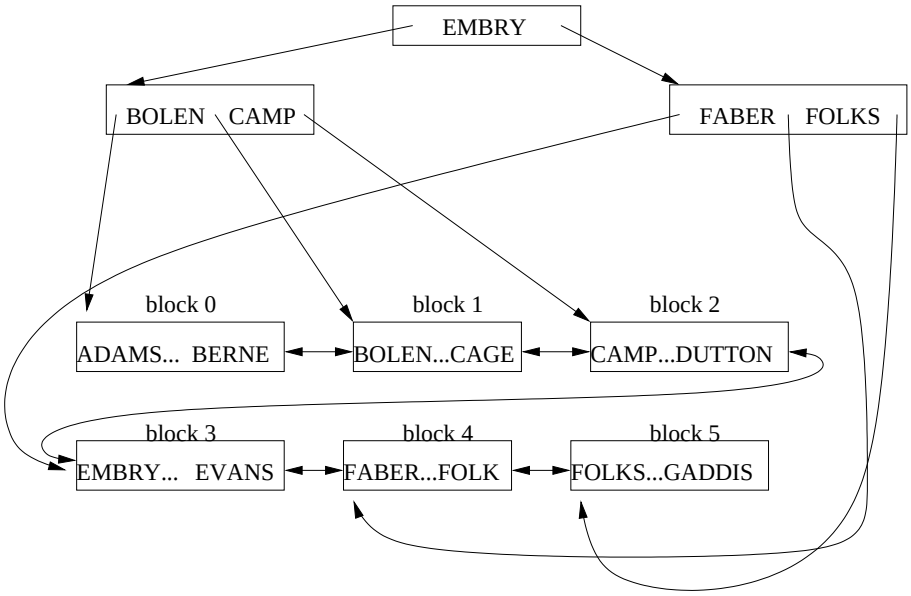

Figura 14: Árvore B^+.

14 Árvore B^+ com prefixo simples

14.1 Motivação

O conceito de separador do tipo **prefixo simples** permite aumentar o número de separadores por nó da árvore B, reduzindo sua altura. O prefixo simples que separa um bloco do anterior é a cadeia de caracteres mais curta que diferencia a última chave do bloco anterior da primeira chave do bloco atual (ver Figura 15).

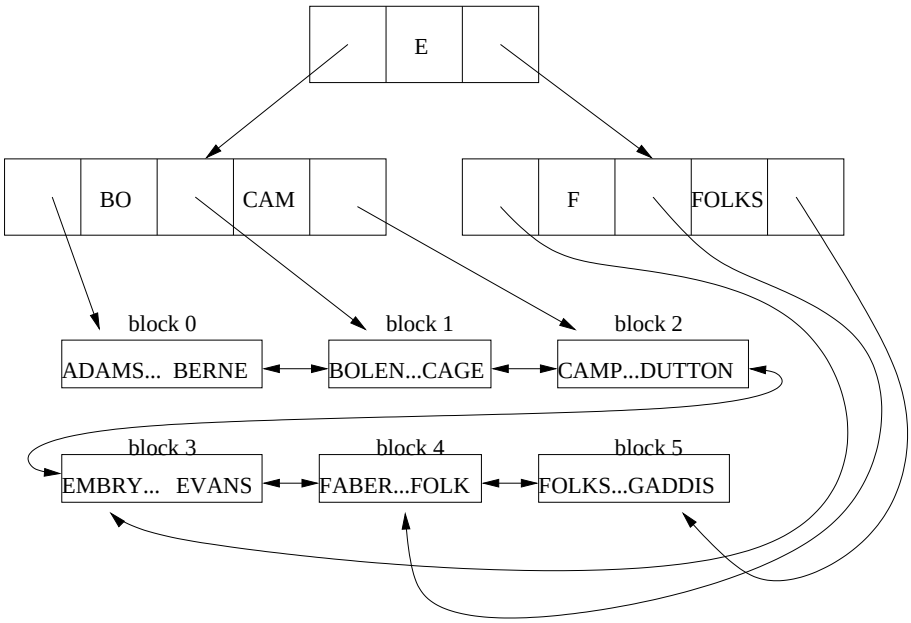

Figura 15: Prefixo simples

14.2 Aspectos sobre inserção e remoção

Considere a árvore B^+ com prefixo simples da Figure 15. Note que se removermos os registros com chaves EMBRY e FOLKS, os separadores continuam válidos (Figure 16). Se a árvore B estiver sendo mantida em disco, o custo de atualizar os separadores pode não compensar. A remoção de EMBRY faz com que o segundo registro, cuja chave é ERVIN, assuma sua posição no conjunto de seqüências, e o separador E continua sendo válido entre DUTTON e ERVIN. A remoção de FOLKS faz com que o segundo registro, cuja chave é FROST, assuma sua posição. O separador FR seria mais adequado entre FOLK e FROST, mas podemos manter FOLKS como separador. Portanto, a existência do separador não implica na existência da chave no conjunto de seqüências. O mesmo exemplo vale para casos de inserção sem divisão.

Por outro lado, inserções que geram divisão e remoções que geram união ou redistribuição, são normalmente acompanhadas da atualização da árvore B. A Figura 17 ilustra o caso para inserção com divisão de um registro e a Figura 18 ilustra o caso para remoção com união. No primeiro caso, o separador AY deve ser inserido na árvore B, e o separador CAM deve ser removido no segundo caso.

14.3 Estrutura dos nós da árvore B

Cada nó da árvore bem como os elementos do conjunto de seqüências são blocos de tamanho fixo. Para facilitar a indexação em RBN, os blocos devem ter o mesmo tamanho em ambas estruturas, porém os separadores são de tamanho variável. Em vez de vetores de separadores, usa-se uma única cadeia de caracteres para armazenar os separadores e um vetor de índices

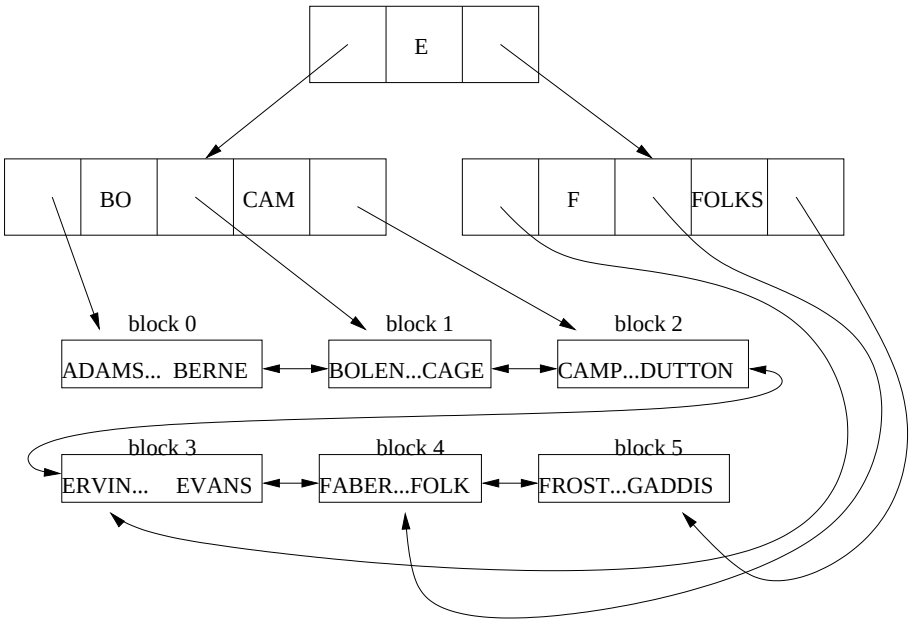

Figura 16: Remoção na árvore da Figura 15 que não requer atualizar os separadores da Árvore B.

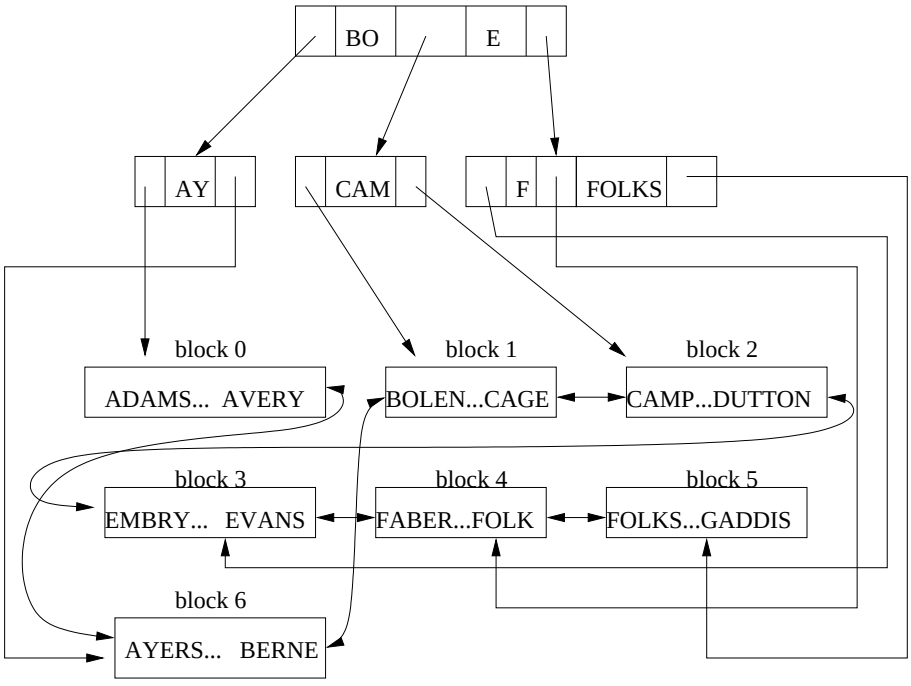

Figura 17: Inserção na árvore da Figura 15 que requer atualização dos separadores da Árvore B.

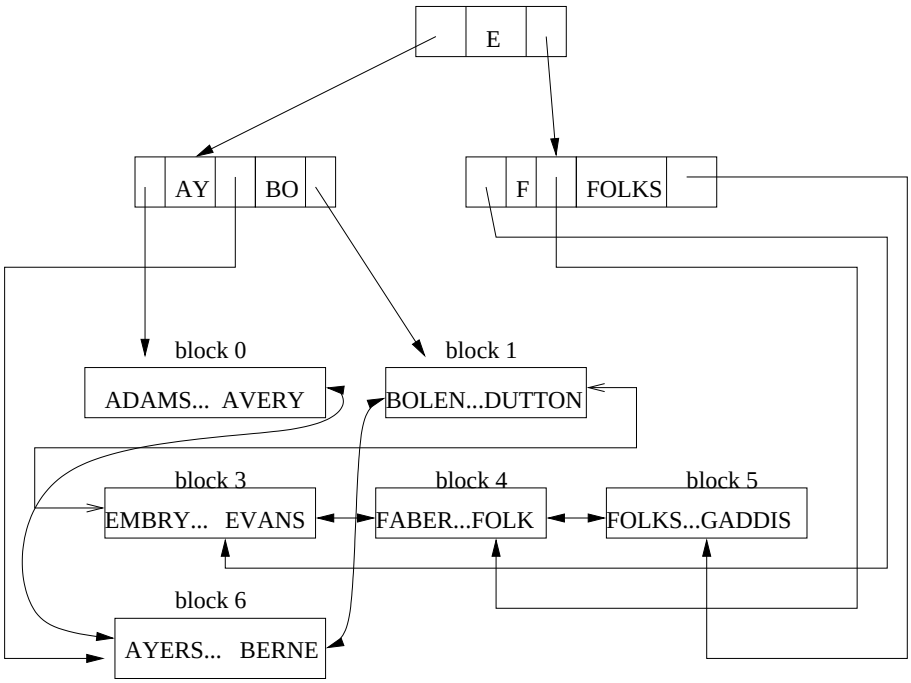

Figura 18: Remoção na árvore da Figura 17 que requer atualização dos separadores da Árvore B.

inteiros com a posição relativa dos separadores. O número atual de separadores, o comprimento da cadeia de separadores e o vetor de endereços dos filhos também são armazenados (ver Figura 19). Observe que a busca binária de um separador é feita através do vetor de índices, e como a única limitação é o tamanho do bloco, a árvore B tem **ordem variável** por nó. Isto implica em modificações nas rotinas originais.

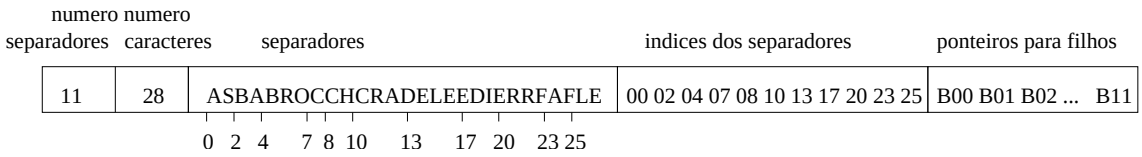

Figura 19: Nó da árvore B de separadores.

14.4 Geração de uma árvore B$^+$ com prefixo simples

A forma mais eficiente de gerar uma árvore B$^+$ com prefixo simples é primeiro ordenar os registros pela chave criando o conjunto de seqüências em disco, e depois inserir um a um os separadores ordenados na árvore B. À medida que transitamos de um bloco para outro do conjunto de seqüências, calculamos o separador e inserimos ele na árvore B. Os blocos da árvore

B permanecem em memória principal e só são gravados em disco no instante de uma divisão (ver Figura 20). A divisão gera dois novos blocos na memória RAM, um sem separadores no mesmo nível e outro raíz com o separador recém-inserido. Ao final, se sobrar um bloco sem separadores, nós redistribuimos os separadores entre os blocos irmãos.

Esta abordagem é mais eficiente do que a normalmente usada para criar uma árvore B. No caso tradicional, os blocos ficam 67% a 80% cheios. Nesta abordagem, eles ficam cheios e a árvore B mais compacta.

Uma alternativa para agilizar o acesso a disco é ler o conjunto de seqüências ordenado, copiando os seus registros para um novo conjunto e, ao mesmo tempo, inserindo os separadores na árvore B, gravando os nós da árvore no mesmo arquivo que o conjunto. Esta alternativa faz com que os endereços dos nós da árvore e dos blocos correspondentes no conjunto fiquem mais próximos.

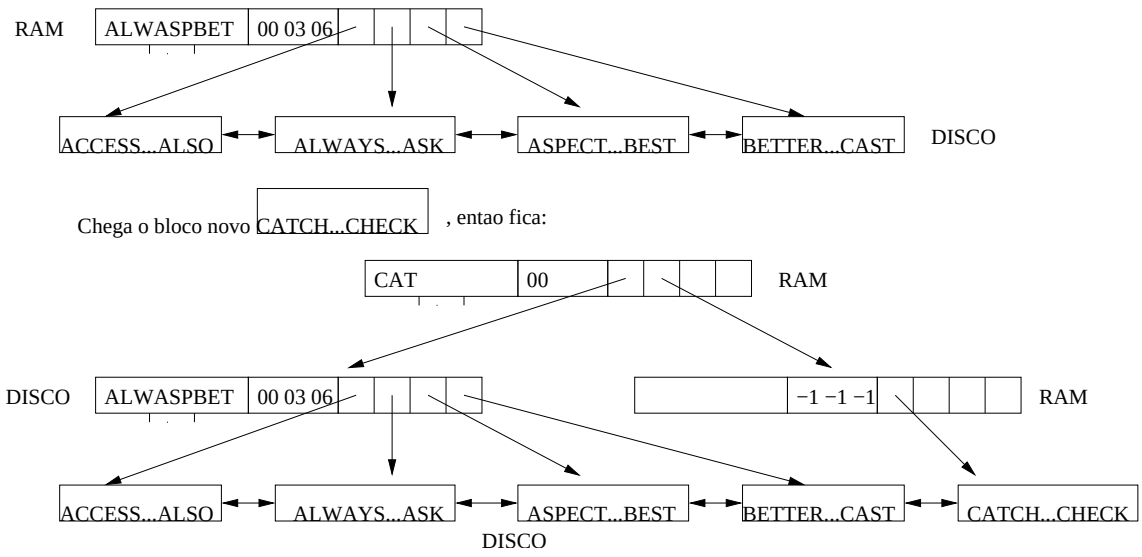

Figura 20: Geração da árvore B onde os nós só são gravados em disco quando se dividem.

15 Espalhamento (Hashing) em disco

Seja r o número total de registros a serem armazenados em um índice (primário ou secundário), o qual está dividido em blocos de tamanho fixo com capacidade para armazenar um certo número n de registros de tamanho fixo. Nosso objetivo é fazer acesso a disco em $O(1)$ usando a técnica de espalhamento. Neste caso, uma função de espalhamento leva uma chave k (primária ou secundária) em um endereço $h(k)$ do arquivo em disco, dado em RBN.

As chaves mapeadas em um mesmo bloco são denominadas **sinônimas** e uma **colisão** ocorre quando o número de chaves mapeadas em um bloco é maior que n. Técnicas de espalhamento visam essencialmente minimizar o número de colisões distribuindo os registros de forma mais

uniforme possível ao longo do arquivo. Entretanto, colisões sempre ocorrem e devem ser resolvidas usando técnicas de tratamento de colisões.

15.1 Funções de espalhamento

Considere $n = 1$ e N o número máximo de endereços disponíveis (i.e. total de blocos) no índice. Isto é, o número médio ideal de registros por bloco seria $n = 1$. Para se aproximar deste objetivo, uma função de espalhamento deve satisfazer algumas características.

Uma função de espalhamento deve buscar sempre uma representação numérica para as chaves. Por exemplo, $k = LOWELL$ pode ser codificada em ASCII como $L = 76$, $O = 79$, $W = 87$, $E = 69$, $L = 76$, $L = 76$. Se as chaves forem representadas com 12 caracteres, sendo 32 o código do caracter espaço em branco, teremos o número 767987697676323232323232 representando LOWELL. Obviamente, este valor é muito alto, mas pode ser reduzido se somarmos os caracteres dois a dois:

```
7679 + 8769 + 7676 + 3232 + 3232 + 3232 = 30820.
```

Porém, em algumas arquiteturas o inteiro máximo é 32767 e não queremos que nenhuma soma de pares de caracteres ultrapasse este valor. Observando que $ZZ = 9090$ é o maior valor somado, podemos limitar o valor da chave escolhendo um número primo (e.g. 19937) que somado ao 9090 não ultrapasse 32767 (e.g. $19937 + 9090 < 32767$), e limitando as somas parciais a este valor:

```
(7679  + 8769) % 19937 = 16448
(16448 + 7676) % 19937 = 4187
(4187  + 3232) % 19937 = 7419
(7419  + 3232) % 19937 = 10651
(10651 + 3232) % 19937 = 13883.
```

A escolha de números primos favorece a distribuição uniforme. Devemos também garantir que o endereço final $h(k)$ não ultrapasse o número de endereços do arquivo. Esta limitação também pode ser feita com o resto da divisão por um número primo (e.g. $h(k) = 13883\%101 = 84$).

Uma questão importante é dimensionar o índice visando minimizar a probabilidade de colisões. Por exemplo, se reservarmos $N = 100$ endereços de memória (1 registro por endereço) e o espalhamento gerar $r = 10000$ endereços, teremos muitas colisões. Podemos então dimensionar o índice, prevendo o número de colisões.

Suponha que, A e B são eventos indicando que um dado endereço não foi escolhido e que um dado endereço foi escolhido, respectivamente. A probabilidade $P(B) = b = \frac{1}{N}$ na distribuição uniforme. Portanto, $P(A) = a = 1 - b$. A probabilidade deste endereço ser escolhido duas vezes é $P(BB) = b^2$, e de uma seqüência $BAABB$ de eventos é $P(BAABB) = a^2b^3$. Ou seja, se em r espalhamentos temos x B's e $r - x$ A's, a probabilidade de uma seqüência é $a^{(r-x)}b^x$ e o número de possibilidades disto ocorrer é $c = \frac{r!}{(r-x)!x!}$. Então, a probabilidade $p(x)$ de um dado endereço ser escolhido x vezes e não ser escolhido $r - x$ vezes é

$$p(x) = \frac{r!}{(r-x)!x!}\left(1 - \frac{1}{N}\right)^{r-x}\left(\frac{1}{N}\right)^{x}. \quad (2)$$

O custo computacional desta função é alto, mas pode ser reduzido usando uma distribuição de Poisson como aproximação. Portanto, usaremos

$$p(x) = \frac{(r/N)^x \exp(-r/N)}{x!} \quad (3)$$

Por exemplo: para $r = N = 1000$, $p(0) = 0.368$, $p(1) = 0.368$, $p(2) = 0.184$ (colisão de 1 registro), $p(3) = 0.061$ (colisão de 2 registros), etc.

Note que, $Np(x)$ é o número esperado de endereços com x registros cada. Então, para $r = N = 1000$, devemos ter 368 endereços sem registros, 368 endereços com 1 registro cada, 184 endereços com 1 colisão cada, 61 endereços com 2 colisões cada, etc. A menos que os 368 endereços sem registros sejam usados para tratamento de colisões, eles representam disperdício de memória. O número total de colisões pode ser previsto como $1Np(2) + 2Np(3) + 3Np(4) + 4Np(5) + \ldots = 363$ (i.e. $36,3\%$). Este número pode ser reduzido se aumentarmos o número de endereços disponíveis, i.e. $N > r$, o que define uma razão r/N denominada **densidade de empacotamento**. Por exemplo, $r = 500$, $N = 1000$, gera $1Np(2) + 2Np(3) + 3Np(4) + 4Np(5) + \ldots = 107$. Neste caso, o percentual de registros em colisão (*overflow*) é $107/500 = 21.4\%$.

Portanto, para 50% de densidade de empacotamento com 1 registro por bloco teremos cerca de 21% dos registros armazenados em endereços diferentes de $h(k)$. Por outro lado, quanto menor for r/N maior será o disperdício de memória. Note que se adotarmos $n > 1$, a densidade de empacotamento cai para r/nN e uma colisão ocorre apenas quando $x > n$.

15.2 Tratamento de colisões

Analisando as técnicas mais comuns de tratamento de colisões temos que:

- Espalhamento linear (*overflow* progressivo)

  A probabilidade de ocorrer um mapeamento em um dado endereço não é uniforme e aumenta com o número de blocos ocupados antes do endereço. Por exemplo, a probabilidade de ocorrer um mapeamento no bloco 3 será 4 vezes maior que a probabilidade de ocorrer um mapeamento no bloco 0, caso os blocos 0, 1, e 2 já estiverem ocupados.

  A busca por um registro também pode ficar lenta, se este não estiver no seu primeiro endereço de espalhamento (*home address*). Assim, o **comprimento de busca** de um dado registro é definido como o número de acessos a disco necessário para chegar até o registro. Suponha, por exemplo, que para $n = 1$ temos os blocos 0, 1 e 2 ocupados com 1 registro cada e que os endereços dos blocos 0 e 1 são os *home addresses* de seus respectivos registros, mas o registro do bloco 2 tem *home address* no bloco 0. O comprimento de busca deste registro é 3, enquanto o dos outros dois é 1. Neste caso dizemos que o **comprimento médio de busca** (CMB) é $\frac{1+1+3}{3}$ (média dos comprimentos de busca de todos os registros armazenados).

  Quanto maior for r/N, maior será CMB (aumento exponencial). O ideal é manter $r/N < 40\%$. Note que CMB se reduz quando armazenamos $n > 1$ registros por bloco.

Para $n = 1$, $r = 750$ e $N = 1000$, temos por exemplo $r/N = 75\% > 40\%$. Se $n = 2$, a densidade de empacotamento se reduz para $r/(nN) = 37.5\% < 40\%$. O número esperado de colisões também se reduz para um mesmo espaço em disco. Para $n = 1$, $r = 750$ e $N = 1000$, temos $N[1p(2) + 2p(3) + \ldots] = 222$ colisões (i.e. $222/750 = 29.6\%$). Para $n = 2$, $r = 750$ e $N = 500$, temos $N[1p(3) + 2p(4) + \ldots] = 140$ colisões (i.e. $18,7\%$ de *overflow*).

- Espalhamento duplo

  Os registros podem ficar muito distantes no disco, aumentando o tempo de busca.

- Lista encadeada (*overflow* progressivo encadeado)

  Requer gasto de espaço em disco com ponteiros.

- Lista encadeada em arquivo separado (encadeamento em área de *overflow* separada).

  O ideal nas técnicas acima seria evitar que os registros encontrem seus *home addresses* já ocupados por colisões anteriores. Isto requer que todos os registros com *home addresses* diferentes sejam mapeados antes dos demais, o que nem sempre é viável. O uso de um arquivo separado para *overflow* resolve o problema, mas também requer gasto de espaço em disco com ponteiros e aumenta o tempo de busca.

16 Métodos de Acesso Métrico

Os avanços tecnológicos permitem hoje o armazenamento de grandes volumes de **objetos**, tais como imagens, áudio, cadeias de DNA, que não possuem uma relação de ordem natural (i.e. $O_i < O_j$). A ausência desta relação inviabiliza o acesso usando estruturas tradicionais de indexação, tais como árvores B e B^+. Por outro lado, esses objetos possuem uma relação de **similaridade** (i.e. **função distância** $d(O_i, O_j)$) que pode ser explorada para indexação. Neste caso, estamos interessandos em buscas não-exatas onde vários registros satisfazem o critério de busca. Este tipo de busca é denominado **consulta por similaridade**.

Objetos mais complexos também requerem uma **representação** mais simplificada para serem comparados. Normalmente um **vetor de características** do objeto usado para indexação. Por exemplo, cor, forma e textura são propriedades usadas para representar imagens. Consultas aplicadas a uma mesma representação, mas usando funções de distância diferentes, geram resultados diferentes. Portanto, o par representação e função de distância, normalmente uma **métrica**, constitui um **descritor** da coleção de objetos. O descritor reflete a distribuição dos objetos (suas representações) no espaço métrico correspondente. Objetos similares estarão próximos neste espaço, enquanto objetos dissimilares estarão mais afastados. Um descritor com boa **efetividade** deve ser capaz de gerar grupos compactos e bem separados, sendo que os grupos devem ser formados por objetos **relevantes** para consultas a partir de objetos similares a este grupo.

Para tornar **eficiente** a recuperação dos dados, objetos similares devem ser armazenados em blocos próximos no disco. Isto é feito por **árvores métricas** (i.e. estruturas de indexação tais como uma árvore M e seus variantes).

16.1 Consultas por similaridade

Os dois tipos mais comuns de consulta por similaridade são:

- Vizinhos mais próximos: Quais os n objetos mais similares a um dado objeto de consulta.

  Exemplos: Quais as n imagens mais similares a uma imagem de consulta? Quais os n atletas com peso mais próximo do peso do atleta fulano? Quais as n cidades mais próximas de Campinas?

- Abrangência: Quais os objetos cuja similaridade com um dado objeto de consulta é maior que um dado valor.

  Exemplos: Quais as n imagens cuja similaridade com uma imagem de consulta é maior que um dado valor? Quais os atletas cuja altura difere da altura do atleta fulano em no máximo 5cm? Quais as cidades em um raio de 50Km de distância de Campinas?

16.2 Descrição

No caso de objetos mais complexos, uma dificuldade adicional é encontrar um descritor consistente com o tipo de consulta e que resulte em resultados relevantes. Por exemplo, podemos acessar uma base de imagens usando uma imagem de consulta, um desenho sobre seu conteúdo, uma descrição sobre características observadas na imagem. Cada tipo de acesso requer a escolha de uma representação e função de distância adequadas. Se estamos usando um contorno como consulta, representações de cor e textura não têm significado. Precisamos de uma representação de forma que gere resultados de comparação com algum significado para o usuário.

Suponha agora que o objetivo seja recuperar imagens com "paisagens parecidas" a de uma dada imagem de consulta em um banco de imagens de paisagens. Se representarmos as imagens por histogramas de cor, estaremos assumindo que "paisagens parecidas" são aquelas com distribuições percentuais próximas para cada cor. Assim, a imagem de uma laranja pode ser considerada similar a de um sol se pondo, mesmo que isto não faça sentido algum.

Quando a semântica está associada a regiões das imagens, a escolha do descritor fica ainda mais complicada. Por exemplo, suponha que o objetivo seja recuperar informações sobre peixes de uma mesma espécie em um banco de imagens de peixes. A escolha do descritor implica em:

- Definir de forma precisa a extensão espacial dos peixes em todas as imagens da base (**segmentação de imagens**).

- Encontrar uma representação e função de distância capazes de identificar peixes de uma mesma espécie e distinguir peixes de espécies diferentes.

A segmentação automática nem sempre é possível. No pior caso, os peixes devem ser segmentados manualmente antes da inserção de dados na base. A identificação das espécies baseada na forma, cor e/ou textura dos peixes nem sempre é válida. Supondo que este não é o caso, existem ainda vários tipos de representação baseados em forma, cor e/ou textura. Como saber qual é a mais adequada? Esta questão está associada com a efetividade do descritor

(Seção 16.4). Suponha ainda que as espécies podem ser identificadas pela forma. Mais precisamente, pela comparação entre os vetores de curvatura dos pontos ao longo do contorno de cada peixe. Mesmo assim, a função de distância deve levar em conta que peixes de uma mesma espécie podem estar em orientações e tamanho diferentes em ambas as imagens. Funções diferentes levarão a resultados diferentes. Mais uma vez a escolha da função está associada com a efetividade do descritor (Seção 16.4).

Por fim, vamos considerar que encontramos um descritor adequado para o problema. A indexação das imagens em disco (e outros dados associados) usando este descritor deve ser eficiente (rápida). A eficiência da indexação depende da estrutura de dados escolhida (e.g. árvore M ou variantes) e da complexidade da função de distância. Essas estruturas obtêm eficiência explorando propriedades métricas da função de distância. Ou seja, a função pode ser um algoritmo ou uma função matemática (distância de CityBlock, distância Euclideana entre vetores de características), desde que satisfaça as restrições abaixo.

- Positividade: $0 < d(O_i, O_j) < \infty$, se $i \neq j$, e $d(O_i, O_i) = 0$.
- Simetria: $d(O_i, O_j) = d(O_j, O_i)$.
- Desigualdade triangular: $d(O_i, O_j) \leq d(O_i, O_k) + d(O_k, O_j)$.

16.3 Árvores Métricas

16.4 Medidas de efetividade

Medidas de efetividade são usadas para avaliar descritores com relação a relevância dos resultados de uma consulta.

Em algumas situações, o critério de relevância não é fixo, ele depende do usuário. Técnicas como *relevance feedback* permitem que o usuário especifique o grau de relevância de cada objeto recuperado, e o sistema procura se adaptar a esta especificação, modificando seus parâmetros e o resultado da consulta. Este modelo teórico é muito complicado de ser implementado em situações práticas.