Ordenação de arquivos grandes

6897/9895 – Organização e Recuperação de Dados

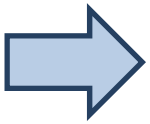
Profa. Valéria D. Feltrim

UEM – CTC – DIN

Slides preparados com base no Cap. 7 do livro FOLK, M.J. & ZOELLICK, B. *File Structures*. 2nd Edition, Addison-Wesley Publishing Company, 1992, e nos slides disponibilizados pelo Prof. Pedro de Azevedo Berger (DCC/UnB)

Ordenação de arquivos grandes

Arquivos que são grandes demais para serem ordenados em RAM



Keysort

- Desvantagens
  - Depois de ordenar as chaves, existe um custo substancial de *seeking* para ler e reescrever cada registro no arquivo novo
  - O tamanho do arquivo é limitado pelo número de pares chave/ponteiro que pode ser armazenado na RAM
    - Inviável para arquivo realmente grandes

Ordenação de arquivos grandes

- Exemplo hipotético:
  - Características do arquivo a ser ordenado:
    - 800.000 registros
    - Tamanho do reg: 100 bytes
    - Tamanho da chave: 10 bytes
  - Tamanho total do arquivo: 80 MB
  - Memória disponível para a ordenação: 1 MB
  - Número total de bytes só para as chaves: 8 MB

Não é possível fazer *Ordenação interna* nem *Keysorting*

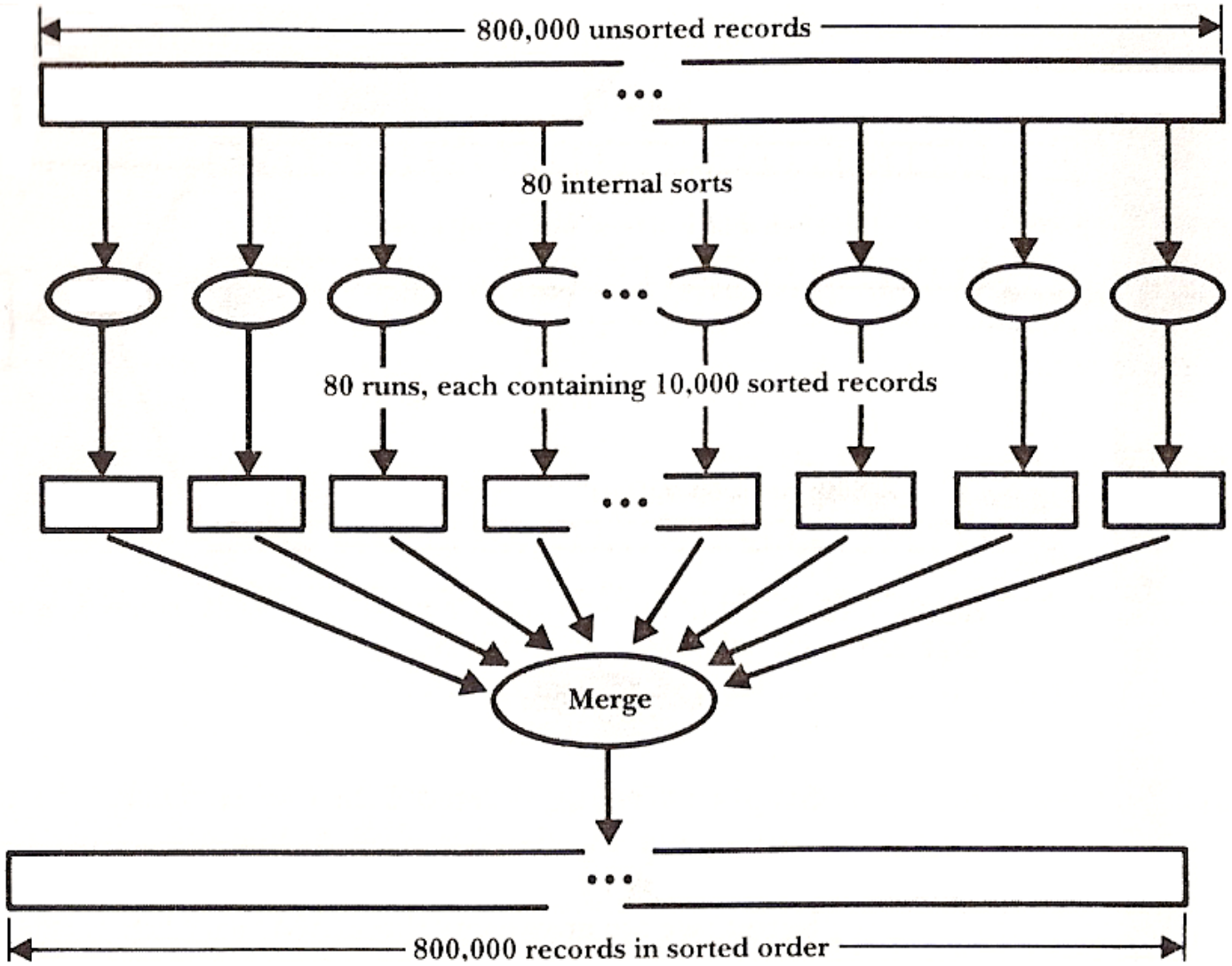
Ordenação de arquivos grandes

- *Merge Sort*
  1) Criar arquivos menores ordenados chamados de "*runs*":
     - Trazer o máximo de registros possíveis para a memória, fazer **ordenação interna** e **salvar em um arquivo temporário (*run*)**
       - Qualquer algoritmo de ordenação interna pode ser utilizado na criação das *runs*
     - Repetir o processo até que todos os registros do arquivo original tenham sido lidos
  2) Fazer intercalação múltipla (***K-way merge***) dos sub-arquivos ordenados
     - *Merge* em *K*-vias
     - *K* é igual ao número de *runs* que serão intercaladas

Merge Sort

- No exemplo anterior (arquivo de 80 MB), qual seria o tamanho de cada *run*?
  - Memória disponível = 1MB = 1.000.000 bytes
  - Tamanho do registro = 100 bytes
  - Total de registros que cabe na memória disponível?
    - 10.000 registros
  - Se o número total de registros é 800.000, qual o número total de *runs*?
    - 80 *runs*
- As 80 *runs* estarão em 80 arquivos separados → realização de ***80-way merge*** para gerar o arquivo final ordenado

Merge Sort


800,000 unsorted records
80 internal sorts
80 runs, each containing 10,000 sorted records
Merge
800,000 records in sorted order

Merge Sort

- Características do *Merge Sort* (abordagem "criar *runs*" + "intercalação múltipla")
  - É extensível para arquivos de qualquer tamanho
  - A leitura do arquivo de entrada para a criação das *runs* é sequencial
  - A leitura das *runs* durante o processo de *merging* e a escrita dos registros ordenados também é sequencial
    - Acesso aleatório é necessário quando alternamos a leitura de uma *run* para outra durante o *merge*

Quanto tempo custa o *Merge Sort* em disco?

- O maior custo da ordenação externa é devido as operações em disco
- No *Merge Sort*, operações de E/S são realizadas 4 vezes
  - Durante a fase de ordenação (*Sorting*)
    1. Leitura dos registros para a memória para ordenar e criar *runs*
    2. Escrita das *runs* ordenadas no disco
  - Durante a fase de intercalação (*Merging*)
    3. Leitura das *runs* ordenadas do disco para a memória para realizar o *merge*
    4. Escrita do arquivo final ordenado no disco

Quanto tempo custa o *Merge Sort* em disco?

1. Leitura de cada registro para a memória para ordenar e criar as *runs*
2. Escrita das *runs* ordenadas para o disco

- Os passos 1 e 2 são feitos da seguinte forma:
  - Leia um bloco de 1MB e escreva uma *run* ordenada de 1MB
    - 1 seek para a leitura e 1 seek para a escrita
  - Repita o passo anterior 80 vezes
  - Em termos de operações em disco:
    - Para leitura: 80 seeks + tempo de transferência p/ 80MB
    - Para escrita: 80 seeks + tempo de transferência p/ 80MB

Quanto tempo custa o *Merge Sort* em disco?

3. Leitura das *runs* ordenadas do disco para a memória para realizar o *merge*

- Para que o *merge* possa ser realizado, é preciso ler as 80 *runs* simultaneamente
  - Divida a memória de 1MB em 80 *buffers* de entrada
  - Cada *buffer* conterá 1/80 de uma *run*
    - Cada *run* será acessada 80 vezes para que seja lida por completo
    - Consideramos 1 seek para cada acesso
  - Cada uma das 80 *runs* será acessada 80 vezes (80*80 = 6.400)
    - Operações em disco: 6.400 seeks + tempo de transferência de 80MB

Quanto tempo custa o *Merge Sort* em disco?

4. Escrita do arquivo ordenado para o disco

- Para escrever o arquivo ordenado no disco, o número de *seeks* depende do tamanho do *buffer* de saída
  - Qtd de bytes no arquivo/Qtd de bytes no *buffer* de saída
    - Ex.: se o *buffer* de saída for de 20KB serão 4.000 *seeks* (80MB/20KB)
  - A transferência novamente será de 80MB

    Obs.: A memória de 1MB está sendo usada pelo passo 3, de modo que pelo menos um *buffer* de saída adicional será necessário

Conclusão: A fase de *merging* (passos 3 e 4) é o gargalo

Custo do *Merge Sort*

- E se o arquivo que se quer ordenar tiver 800MB em vez de 80MB?
  - Com o mesmo tamanho de memória (1MB)
  - Em que proporção aumentarão os *seeks* e as transferências?

O número de ***seeks*** será **100 vezes maior** para o arquivo de 800MB

Enquanto a entrada foi multiplicada por 10, o número de *seeks* foi multiplicado por 100!
(Teremos **680.000** *seeks* só na fase de *merging*)

Custo do *Merge Sort*

- Em geral, para um ***K-Way Merge*** de ***K runs***, em que cada *run* é do tamanho da memória disponível, o tamanho do *buffer* para cada *run* é de:

  (1/K) * tamanho da MEMÓRIA = (1/K) * tamanho de cada *run*

  - *K* seeks serão necessários para acessar todos os registros de cada *run*
- Como temos *K runs* e cada *run* terá que ser lida *K* vezes, a operação de intercalação requer K^2 *seeks*
  - Assim, medido em termos de *seeks*, o *Merge Sort* é $O(K^2)$
  - Como *K* é diretamente proporcional a *N*, podemos dizer que o ***MergeSort* é $O(N^2)$**, em termos de *seeks*

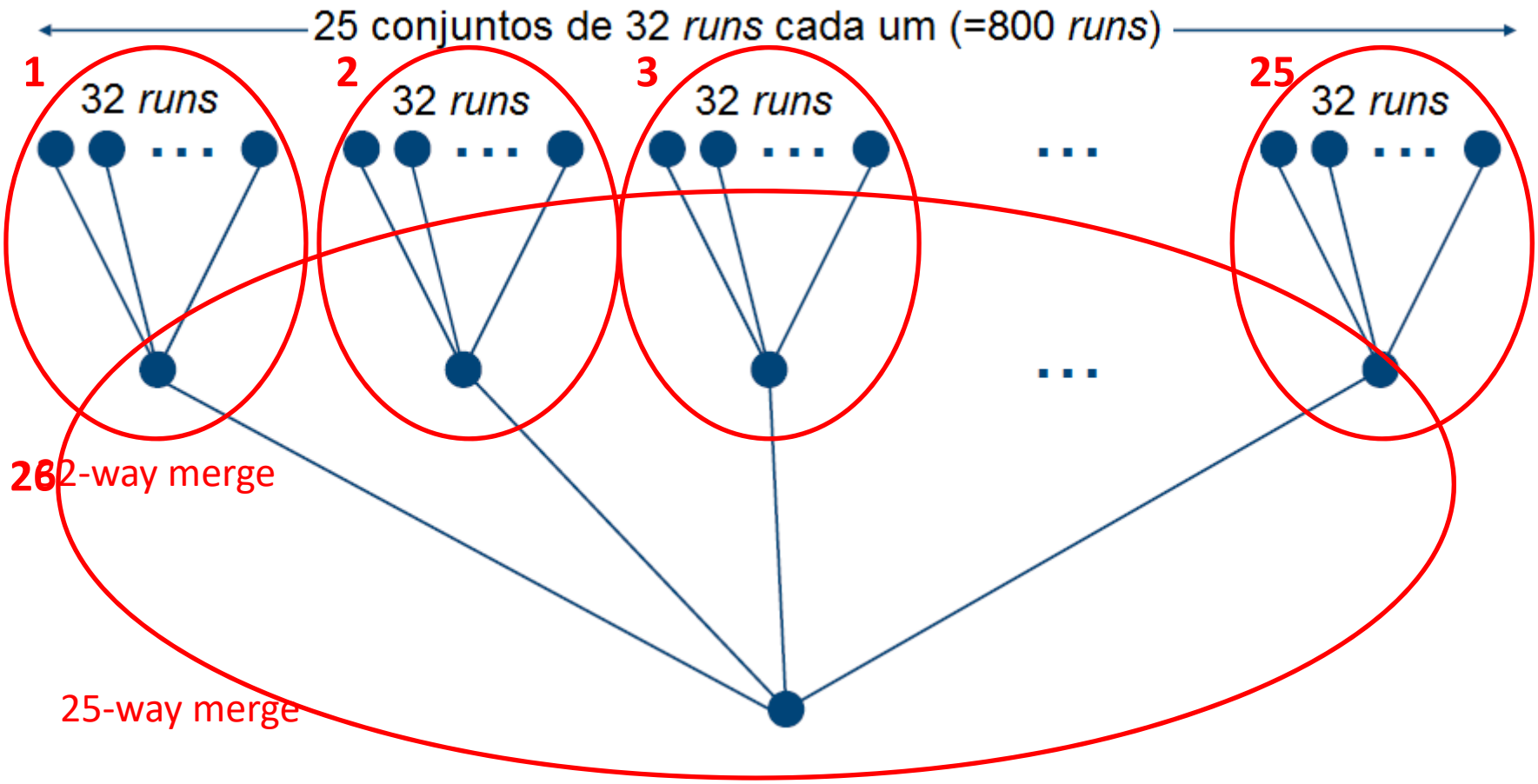
Conforme o arquivo cresce, o tempo requerido para realizar a ordenação cresce rapidamente!

Redução do custo do *Merge Sort*

- Merge em múltiplos passos
  - Em vez de fazer o *merge* de todas as *runs* ao mesmo tempo, o grupo original de *runs* é dividido em sub-grupos menores
  - Um *merge* é feito para cada sub-grupo
    - Para cada um dos sub-grupos, um espaço maior do *buffer* poderá ser alocado para cada *run*, portanto, um número menor de *seeks* será necessário
  - Uma vez completados todos os *merge* "pequenos", o segundo passo completa o *merge* das *runs* resultantes do passo anterior

Merge em múltiplos passos

- Merge em 2 passos para o arquivo de 800MB



Custo do merge em múltiplos passos

- Temos menos *seeks* na primeira passada, mas ainda há uma segunda passada → no final, compensa?
- *800-way merge* original: **640.000 *seeks* no passo 3**
- *Multiple-step merging* (25 x 32-way merge + 25-way merge): **?**
  - 1ª passada:
    - Cada *merge* de 32-vias aloca *buffers* que podem conter 1/32 de uma *run*, então serão realizados 32 *seeks* por *run*
      - 32 *seeks* por *run* * 32 *runs* = 1.024 *seeks* por grupo de *runs*
      - 25 grupos de *runs* * 1.024 *seeks* = **25.600 *seeks***
    - Cada *run* resultante desse passo terá 320.000 registros (32 * 10.000 registros de 100 bytes), o que equivale a um arquivo de 32MB
  - 2ª passada:
    - Cada uma das 25 *runs* de 32MB poderá alocar 1/25 do *buffer* (40.000 bytes). Portanto, cada *buffer* poderá conter 400 registros, o que é igual a 1/800 da *run*
    - 32.000.000 bytes/40.000 bytes = 800 *seeks* por *run*
    - 25 *runs* * 800 *seeks* = **20.000 *seeks***
  - TOTAL = 25.600 + 20.000 = **45.600 seeks**

Merge em múltiplos passos

- Encontramos uma maneira de aumentar o espaço disponível no *buffer* para cada *run*
- Trocamos passadas extras sobre os dados por uma diminuição significativa no acesso aleatório
  - Aumenta o tempo de transferência, mas diminui o número de *seeks*
- Se fizermos um *merge* em 3 passos, podemos obter resultados ainda melhores?
  - Talvez não, pois temos que considerar o tempo de transferência dos dados
    - No *merge* original, os dados são transmitidos 2 vezes (passos 3 e 4) enquanto que no *2-step merge*, os dados são transmitidos 4 vezes
  - Também é preciso considerar a escrita extra dos dados
    - 80.000 *seeks* na escrita dos 1º e 2º merge (com *buffer* de 20K)
    - No total, esse *2-step merge* custou **127.200 *seeks***

Exercício

- Considerando:
  - arquivo em disco de 1 GB
  - *buffer* de entrada (memória RAM) de 5 MB
  - *buffer* de saída de (memória RAM) 500 KB
- calcule o custo em termos de nº de *seeks* e bytes transmitidos do *Merge Sort*

Obs.: faça os cálculos arredondando os valores de tamanho dos arquivos e memória → 1K = 1.000 bytes; 1M = 1.000K; 1G = 1.000M

Exercício

- Considerando:
  - arquivo em disco de 1 GB
  - *buffer* de entrada (memória RAM) de 5 MB
  - *buffer* de saída de (memória RAM) 500 KB
- calcule o custo em termos do nº de *seeks* e bytes transmitidos de cada uma das fases do *Merge Sort* em múltiplos passos, quando realizado em dois passos:
  - 10 x 20-vias + 10-vias

Obs.: faça os cálculos arredondando os valores de tamanho dos arquivos e memória → 1K = 1.000 bytes; 1M = 1.000K; 1G = 1.000M

Merging em Múltiplos Passos

- Observação:
  - Vale notar que as diferenças nos valores das comparações apresentadas nos slides estão exageradas, pois não consideramos o tempo de transmissão
  - Se considerarmos o tempo de transmissão, o *merging* em múltiplos passos continua ganhando, mas as diferenças em termos de tempo serão menores