Ordenação de arquivos grandes

Objetivos

- Mostrar como o merging pode ser utilizado para ordenar arquivos grandes
- Examinar o custo do K-Way em disco e encontrar uma forma de reduzir esse custo
- Introduzir o conceito de *replacement selection*

Ordenação de arquivos grandes

Um problema

- Consider um arquivo com 800.000 registros; cada registro com 100 bytes → Arquivo de 80 MB
- cada registro com uma chave com 10 bytes
- Suponha que temos 1 MB de memória de trabalho (sem contar o programa, sistema operacional, *buffers* de E/S, ..

Para simplificar vamos supor que:

- os arquivos estão armazenados em áreas contíguas do disco (*extents*)
- e o *seek* dentro de um mesmo cilindro (ou de um cilindro para o próximo) não gasta tempo. Portanto, apenas um *seek* é necessário para qualquer acesso seqüencial.
- *extents* alocados em mais que uma trilha são alocados de tal modo que apenas uma unidade de latência rotacional é necessária para cada acesso.

Ordenação de arquivos grandes

Utilizando keysort

- cada registro com uma chave com 10 bytes→espaço para chaves= 8 MB.
- Se temos 1 MB de memória de trabalho não é possível ordenar em MEMÓRIA, mesmo usando o *keysort*.

Utilizando k-vias (*k-way merging*)

Considerando que algoritmos tais como o *heapsort*, podem ser utilizados para trabalhar tanto em MEMÓRIA como em disco a um custo baixo, é possível criar um subconjunto ordenado do arquivo através do seguinte processo:

- ler registros para a MEMÓRIA até lotar
- ordenar esses registros na MEMÓRIA (ordenação interna)
- escrever os registros ordenados em um sub-arquivo (ordenado).
  Cada sub-arquivo ordenado é chamado uma corrida ("*run*")
- **Fazer o merge dos sub-arquivos.**

Ordenação de arquivos grandes

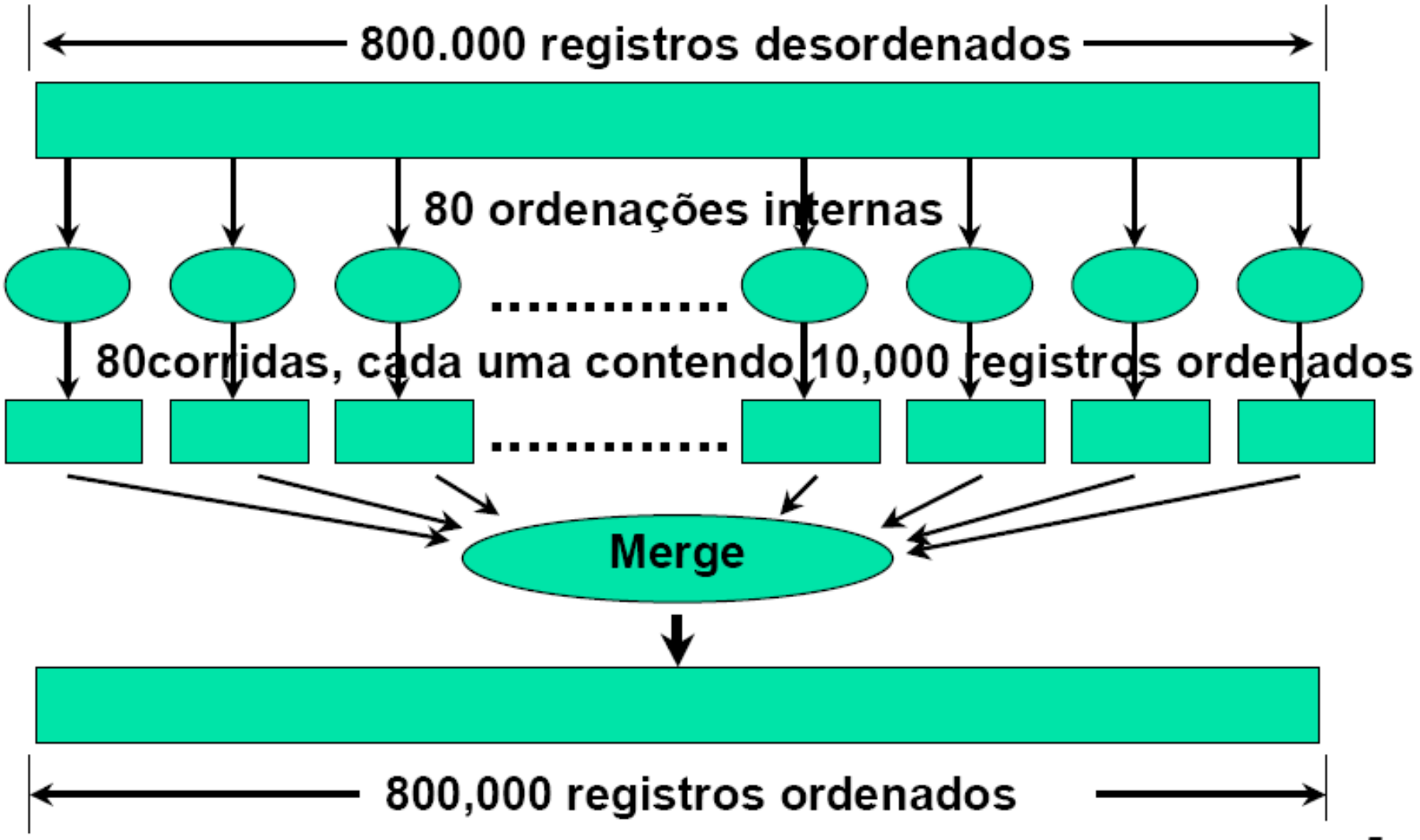
Utilizando k-vias

- Vamos criar corridas com 10.000 registros.
  Como cada registro tem 100bytes →cada corrida tem 1 milhão de bytes
- Uma vez terminada a primeira corrida, repete-se a operação para o resto do arquivo, criando um total de 80 corridas, cada uma com 10.000 registros já ordenados.
- Temos, então, 80 corridas em 80 arquivos separados: podemos realizar um merging em 80-vias para criar um arquivo completamente ordenado.

Conclusão

- Desta forma é possível ordenar arquivos realmente grandes.
- a geração das corridas utiliza leitura seqüencial → muito mais rápido.
- as leituras das corridas e a escrita final também são seqüenciais.
- se o *heapsort* é utilizado na parte do merging realizado em MEMÓRIA, o merging pode ser feito simultaneamente à E/S, e a parte do merging em MEMÓRIA não aumenta o tempo total de execução.

Ordenação através de corridas e subsequente merging



Quanto tempo custa o *MergeSort*

Operações de E/S são realizadas, durante o *mergesort*, em 4 oportunidades:

- fase de ordenação:

  1- leitura dos registros para a memória para a criação de corridas e

  2- escrita das corridas ordenadas para o disco

- fase de intercalação:

  3- leitura das corridas para intercalação (mergesort)

  4- escrita do arquivo final em disco

Quanto tempo custa o *MergeSort*

1. leitura dos registros para criação de corridas (no exemplo)

- Lemos 1MB de cada vez, para produzir corridas de 1 MB.
- MEMÓRIA → *buffer* que vai ser utilizado 80 vezes, para formar as 80 corridas.
- O tempo de leitura de cada corrida inclui o tempos de acesso a cada bloco (*seek* + *latência rotacional*) + tempo para transferir cada bloco.
- Da tabela de tempos, temos: **seek = 18ms; latência rotacional =8.3ms; taxa de transferência =1,229 bytes/ms**
- Tempo total para a fase de ordenação
  seek + latência rotacional + tempo de transferência do arquivo de 80MB.

---------- **Exercício** ----------

- Tempo de acesso p/ uma corrida= *seek* + *latência rotacional* =26.3ms.
  p/ 80 corridas: 80 x (seek+*latência rotacional*)= 80X 26.3ms= 2s
- Tempo de transferência: 80 MB a 1.229 bytes/ms → acertando as unidades temos: 80bytes/1.229 bytes/seg=65s
- **Tempo total para a fase de ordenação = 2s + 65s= 67s**

2. escrita das corridas ordenadas para o disco
Idem à leitura. **Total= 67s**

Quanto tempo custa o *MergeSort*

RESUMO

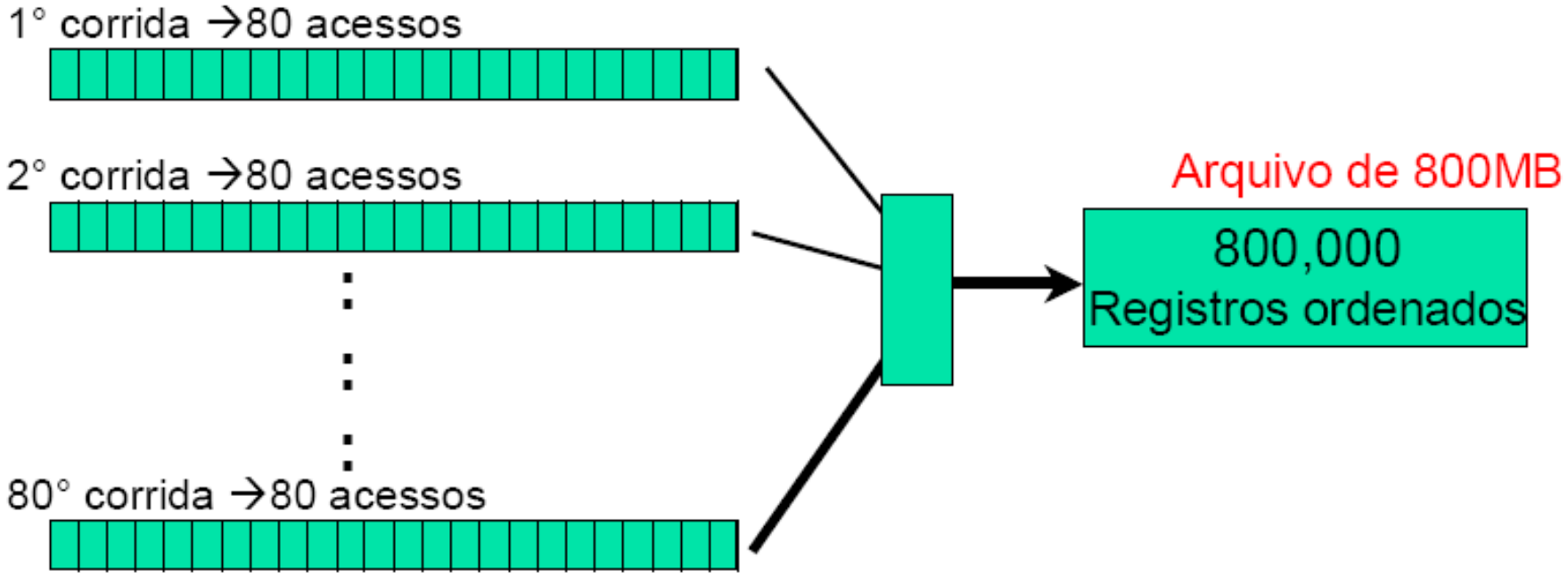
Fase/operação	n.º de seeks	Qdade de transferência (MB)	Seek + latência rot. (s)	Tempo de transf	Tempo total (s)
Ord./ leitura	80	80	2	65	67
Ord./ escrita	80	80	2	65	67
Merge/leitura					
Merge/escrita					
Total					

Quanto tempo custa o *MergeSort*

3- leitura das corridas do disco para a memória (para intercalação)

→ O Efeito da buferização no n° de seeks

- Estamos utilizando 1MB de MEMÓRIA para armazenar 80 *buffers* de entrada, portanto, cada *buffer* armazena 1/80 de uma corrida. Logo, cada corrida deve fazer 80 acessos para ser lida por por completo



Quanto tempo custa o *MergeSort*

3- leitura das corridas do disco para a memória

---------- **Exercício** ----------

- Como precisamos de 80 acessos para ler cada corrida completa, e temos 80 corridas realizamos 80 X 80 *seeks* = 6.400 *seeks*;
- Considerando o tempo de cada *seek* + *latência rotacional* temos:
  6400 X 26.3ms = 168s
  Tempo para transferir 80 MB = 65s (do slide anterior)

  Total=168s + 65s = 233s

Fase/operação	n.º de seeks	Qdade de trans (MB)	Seek + latência rot. (s)	Tempo de transf	Tempo total (s)
Ord./ leitura	80	80	2	65	67
Ord./ escrita	80	80	2	65	67
Merge/leitura	**6.400**	**80**	**168**	**65**	**233**
Merge/escrita					
Total					

Quanto tempo custa o *MergeSort*

4- escrita do arquivo final em disco

- Para calcular o tempo de escrita, precisamos saber o tamanho dos *buffers* de saída. Nos passos 1 e 2 a MEMÓRIA funcionou como *buffer*, mas agora a MEMÓRIA está armazenando os dados a serem intercalados.
- Para simplificar, vamos assumir que é possível alocar 2 *buffers* (adicionais) de 20.000 bytes para escrita (para *buferização dupla*).

----------**Exercício** ----------

- Precisaremos de 80.000.000 bytes/20.000 bytes = 4.000 *seeks* (um acesso sempre que o buffer de saída enche)
- Como o tempo de *seek+latência rotacional* = 23.6ms por *seek*,
  - total de (*seek+ latência rotacional*)= 4.000 X 26.3ms = 105s.
  - Tempo de transferência ainda é = 65s

TOTAL=170s

Quanto tempo custa o *MergeSort*

RESUMO

Fase/operação	n.º de seeks	Qdade de transferência (MB)	Seek + latência rot. (s)	Tempo de transf	Tempo total (s)
Ord./ leitura	80	80	2	65	67
Ord./ escrita	80	80	2	65	67
Merge/leitura	6.400	80	168	65	233
Merge/escrita	4.000	80	105	65	170
Total	10.560	320	277	260	537

8min e 57 s

Quanto tempo custa o *MergeSort*

Comparação: Quanto tempo levaria um método que não usa merging?

- algoritmo *keysort*

---------- *Exercício* ----------

Requer um *seek* separado para cada registro, i.e, 800.000 *seeks* a 26.3ms cada.
tempo total para seek = 21.040s = 5 horas e 40s

- O método *mergesort* é o melhor para arquivos grandes. Mas os resultados podem ser melhorados ainda mais.

Ordenação de um arquivo muito maior

Análise - arquivo de 800 MB (8.000.000 registros com 100 bytes cada)

- O arquivo aumenta, mas a memória não. Em vez de 80 corridas, teremos 800. Portanto, é necessário uma intercalação em 800-vias no mesmo 1 MB de memória , o que implica em que a memória seja dividida em 800 *buffers* na fase de intercalação.
- Na fase de intercalação, cada *buffer* comporta 1/800 de uma corrida, e cada corrida é acessada 800 vezes (ou seja, são necessários 800 *seeks* por corrida).

---------- Exercício ----------

Fase de Merge

- Leitura:total de seeks → 800 corridas X 800 *seeks*/corrida = 640.000 seeks
  seek + latência rotacional = 640.000 X 26.3ms = 16.832s
  Tempo para transferir 800 MB= 800 / 1.229 = 651s
  Tempo=17. 483s
- Escrita: total de seeks=800.000.000 bytes/20.000 bytes = 40.000 *seeks*
  total de (*seek+ latência rotacional*)= 40.000 X 26.3ms = 1050s.
  Tempo de transferência ainda é = 651s

Ordenação de um arquivo muito maior
Cálculos para arquivo de 800MB

Fase/operação	n.º de seeks	Qdade de transf. (MB)	Seek + latência rot. (s)	Tempo de transf	Tempo total (s)
Merge/leitura	640.000	800	16.832	651	17.483
Merge/escrita	40.000	800	1.050	651	1.703
Total	680.000	1.600	17.882	1.302	19.186

5horas, 19min e 22s

- Tempo total para a fase de merge é aproximadamente 36 vezes maior que o arquivo de 80 MB (que é 10 apenas vezes menor).
- **Definitivamente:** é necessário procurar métodos para diminuir o tempo gasto com a obtenção de dados na fase de intercalação.

Ordenação de um arquivo muito maior

O custo de aumentar o tamanho do arquivo

- A grande diferença de tempo na intercalação dos dois arquivos (de 80 e 800 MB) é devida à diferença nos tempos de seek e latência rotacional.

- **Conforme o arquivo cresce, o tempo requerido para realizar a ordenação cresce rapidamente → Maneiras de reduzir esse tempo:**
  - **Usar mais hardware (*disk drives*, MEMÓRIA, canais de I/O)**
  - **Realizar *Multistep Merging***
    - **aumentar o tamanho das corridas iniciais ordenadas**
    - **achar meios de realizar I/O simultâneo**

Melhorias baseadas em hardware

Aumentar o tamanho da memória

- quando dividimos os espaço disponível em *buffers* pequenos, aumentamos os tempos de seek e *latência rotacional.*
- mais memória implica em um menor número de corridas na fase de ordenação; e menor número de *seeks* na fase de intercalação.
- No exemplo, o arquivo de 8.000.000 registros gastou 5 horas e 20 minutos usando 1 MB de MEMÓRIA.
  Se usarmos 4 MB, cada corrida inicial passaria a ter 40.000 registros (em vez de 10.000), e teríamos então 200 corridas de 40.000 registros.
- Na fase de intercalação, a memória seria dividida em 200 buffers, cada um capaz de armazenar 1/200 de uma corrida, o que causaria um total de 200X200 = 40.000 *seeks* (em vez de 640.000)
  tempo → 56 min, 45 sec.

Diminuição do Número de *Seeks* usando *Multistep Merging*

- **Em vez de fazer a intercalação de todas as corridas ao mesmo tempo, o grupo original é dividido em grupos menores**
- **a intercalação é feita para cada sub-grupo.**
- **para cada um desses sub-grupos, um espaço maior é alocado para cada corrida, portanto um número menor de *seeks* é necessário.**
- **uma vez completadas todas as intercalações pequenas, o segundo passo completa a intercalação de todas as corridas.**
- **O segundo passo exige não apenas *seeking*, mas também transferências nas leituras/escritas. Será que as vantagens superam os custos?**

Principais custos do mergsort:

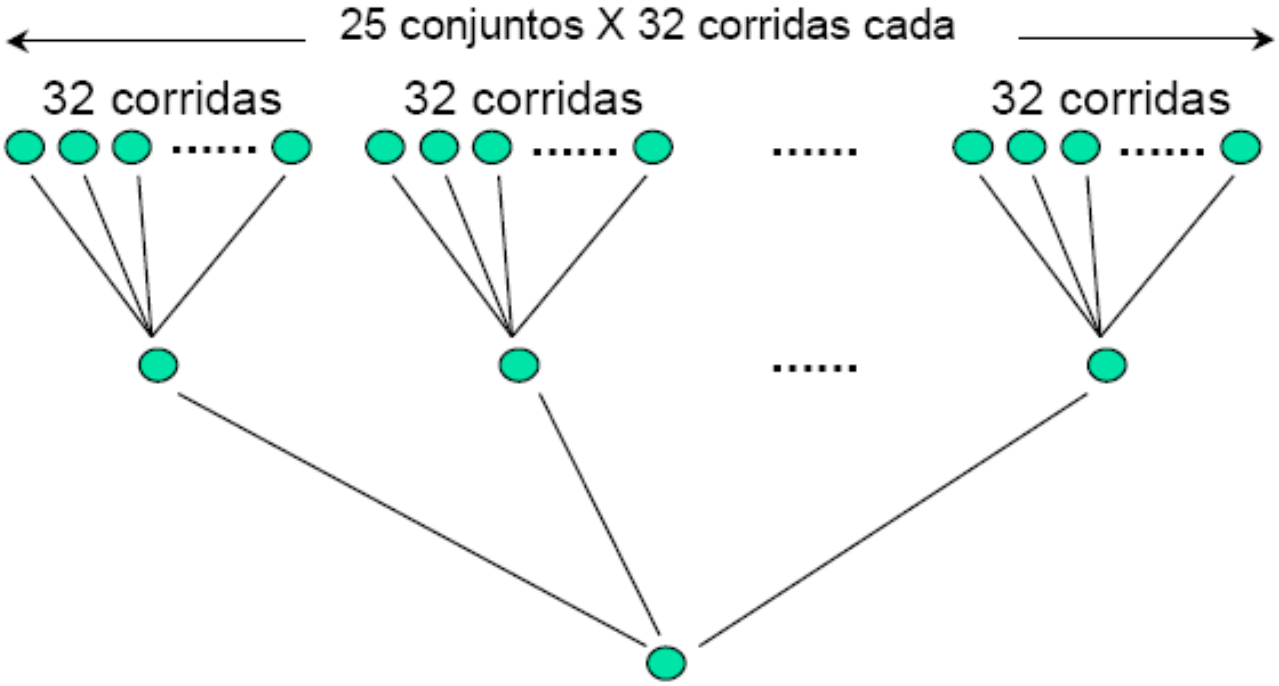
- seek, tempo de transmissão, tamanho do buffer, número de corridas

Diminuição do Número de *Seeks* usando *Multistep Merging*

No exemplo do arquivo com 800 MB tínhamos 800 corridas com 10.000 registros cada. Para esse arquivo, a intercalação poderia ser realizada em dois passos:

1- intercalação de 25 conjuntos de 32 corridas cada

2-intercalação em 25-vias.



Diminuição do Número de *Seeks* usando *Multistep Merging*

- O caso da intercalação em passo único
  - temos 640.000 *seeks* no arquivo de entrada.
- Para a intercalação em 2 passos, temos que cada registro é lido duas vezes, mas o uso de buffers maiores diminui o *seeking*.

Primeiro passo: ---------- Exercício ----------

- Cada intercalação em 32-vias aloca *buffers* que podem conter 1/32 de uma corrida, então serão realizados 32 X 32 = 1024 *seeks*.
- 25 vezes a intercalação em 32-vias exige 25 X 1024 = 25.600 *seeks*
- Cada uma das 25 corridas tem 32 X 10.000 = 320.000 registros = 32 MB.

Segundo passo: ---------- Exercício ----------

- Cada uma das 25 corridas de 32 MB pode alocar 1/25 do *buffer*, portanto, cada *buffer* pode alocar 400 registros.
  Esse passo exige 800 *seeks* por corrida(32.000.000/400reg x100bytes=800)
  total = 25 X 800 = 20.000 *seeks*

- Total de seeks nos dois passos: 25.600 + 20.000 = 45.600
  Ou seja, ao pagarmos o preço de ler cada registro 2 vezes, reduzimos o número de *seeks* de 640.000 para 45.600, sem gastar com memória adicional

Diminuição do Número de *Seeks* usando *Multistep Merging*

E o tempo total de intercalação?

Neste caso, cada registro deve ser transmitido 4 vezes, em vez de duas, portanto gastamos mais 651s em tempo de transmissão.

- **Ainda, cada registro é escrito duas vezes: mais 40.000 *seeks* (assumindo 2 *buffers* com 20.000 posições cada).**
  Somando tudo isso, o tempo total de intercalação = 5.907s ~ 1hora 38 min.
- **Lembre-se do tempo para a intercalação implementada com fase única: 5h 20m**
- **O que fizemos foi aumentar o tamanho do *buffer* disponível para cada corrida. A custo de passos extras, diminuiu-se o número de acessos aleatórios.**
- **Seria possível melhorar o tempo total se utilizarmos 3 passos?**
  Talvez, mas os ganhos provavelmente serão menores, pois no caso de 2 passos, o tempo total de *seek* + *latência rotacional* já foi aproximadamente reduzido ao mesmo que o tempo de transmissão.
- **E se as corridas fossem distribuídas diferentemente? Por exemplo, como seria se fossem realizadas primeiro 400 intercalações em 2-vias, seguida de uma intercalação em 400-vias?**

Replacement Selection

Idéia básica:

- **selecionar na memória a menor chave,**
- **escrever essa chave no arquivo de saída, e**
- **usar seu lugar (replace it) para uma nova chave (da lista de entrada).**

Replacement Selection

Os passos são:

1. **Leia um conjunto de registros e ordene-os utilizando *heapsort*, criando uma heap; a essa *heap* dê o nome de *primary heap*.**
2. **Ao invés de escrever, neste momento, a *primary heap* inteira ordenadamente e gerar uma corrida (como seria feito no *heapsort* normal), escreva apenas o registro com menor chave.**
3. **Busque um novo registro no arquivo de entrada e compare sua chave com a chave que acabou de ser escrita.**
   - **Se chave lida for maior que a já chave escrita, insira o registro normalmente na *heap* → novo registro é parte da corrida sendo criada**
   - **Se a chave lida for menor que qualquer chave já escrita, insira o registro numa *secondary heap*, a qual contém registros menores dos que os que já foram escritos (usando a mesma memória)**
4. **Repita o passo 3 enquanto existirem registros a serem lidos. Ficando a *primary heap* vazia, transforme a secondary heap em *primary heap*, e repita passos 2 e 3.**

Exemplo: Replacement Selection

Input:
21, 67, 12, 5, 47, 16

Início da string

Input restante	**Memoria**(p=3)	**Saída da corrida**
21, 67, 12	5 47 16	-
21, 67	12 47 16	5
21	67 47 16	12, 5
-	67 47 21	16, 12, 5
-	67 47 -	21, 16, 12, 5
-	67 - -	47, 21, 16, 12, 5
-	- - -	67, 47, 21, 16, 12, 5

Exercício

Mostre passo a passo as operações do Replacement Selection com dois heaps de maneira a formar duas corridas ordenadas e mostrar como ficam a memória e a saída de cada corrida

Entrada

33, 18, 24, 58, 14, 17, 7, 21, 67, 12, 5, 47, 16

↑ **Início da string**

Restante da entrada	**Memoria**(P=3)	**Saída da corrida(A)**
33, 18, 24, 58, 14, 17, 7, 21, 67, 12	5 47 16	-
33, 18, 24, 58, 14, 17, 7, 21, 67	12 47 16	5
33, 18, 24, 58, 14, 17, 7, 21	67 47 16	12, 5
33, 18, 24, 58, 14, 17, 7	67 47 21	16, 12, 5
33, 18, 24, 58, 14, 17	67 47 (7)	21, 16, 12, 5
33, 18, 24, 58, 14	67 (17) (7)	47, 21, 16, 12, 5
33, 18, 24, 58	(14) (17) (7)	67, 47, 21, 16, 12, 5

Exercício

mostre passo a passo as operações do Replacement Selection com dois heaps de maneira a formar duas corridas ordenadas e mostrar como ficam a memória e a saída de cada corrida

Primeira corrida completa→ início da segunda

Restante da entrada	Memoria(p=3)	Saída da corrida(B)
33, 18, 24, 58	14 17 7	-
33, 18, 24	14 17 58	7
33, 18	24 17 58	14, 7
-	24 18 58	17, 14, 7
-	24 33 58	18, 17, 14, 7
-	- 33 58	24, 18, 17, 14, 7
	- - 58	33, 24, 18, 17, 14, 7
	-	58, 33, 24, 18, 17, 14, 7

Replacement Selection

- **Dadas P posições de memória, qual o tamanho, em média, da corrida que o algoritmo de replacement selection vai produzir?**

  Resposta:

  - **A corrida terá, em média, tamanho 2P.**
  - **Precisaremos de buffers p/ I/O e, portanto, não poderemos utilizar toda a MEMÓRIA disponível para ordenação.**
  - **Por exemplo, se temos 1 MB de memória reservar um *buffer* de I/O com capacidade de 2.500 registros(de 100 bytes), deixando 7.500 vagas para o processo de *replacement selection*.**

Replacement Selection

Custos - Exercício

- Se o *buffer* suporta 2.500 registros, podemos executar leituras seqüencias de 2.500 registros de cada vez, e portanto precisamos de 8.000.000/2.500 = 3.200 *seeks* para acessar todos os registros do arquivo.
  Portanto, o passo de ordenação requer 3.200 *seeks* para leitura, 3.200 para escrita, 6.400 total.
- Se os registros ocorrem em seqüência aleatória de chaves, o tamanho médio da corrida será 2 X 7.500 = 15.000 registros, e teremos ~ 8.000.000/15.000 = 534 dessas corridas sendo produzidas.
- Para o passo de intercalação, dividimos a memória total (1 MB) em 534 *buffers* que podem conter cerca de 10 MB/534 = 18.73 registros.
  Portanto, realizaremos cerca de 15.000/18.73 = 801 seeks por corrida e, no total: 831 seeks por corrida X 534 corridas = 427.734 *seeks*.

Pontos Básicos para Ordenação Externa

Ferramentas conceituais para melhorar ordenação externa inclui:

- **Para ordenação interna, em-MEMÓRIA, use heapsort para formar corridas. Com heapsort e dupla buferização, é possível sobrepor processamento com E/S.**
- **Use o máximo de MEMÓRIA possível. Isso permite corridas maiores e *buffers* maiores na fase de intercalação.**
- **Se o número de corridas é muito grande, de modo que o *seek* e tempo rotacional são maiores que o tempo de transmissão, use intercalação em múltiplos passos: aumenta o tempo de transmissão, mas diminui em muito o número de *seeks*.**
- **Considere utilizar *Replacement Selection* para formação da corrida inicial, especialmente se existe a possibilidade das corridas estarem parcialmente ordenadas.**
- **Utilize mais de um drive de disco para sobrepor E/S e processamento. Principalmente se não existem outros usuários no sistema.**
- **Lembre-se dos elementos fundamentais da ordenação externa, e os seus custos relativos. E procure maneiras de tirar vantagens das novas arquiteturas e de redes locais de alta velocidade**