SCC 603 – Algoritmos e Estruturas de Dados II – 2012

Processamento Cosequencial*

Professora Rosane Minghim

PAE 2012: Rafael M. Martins

*Baseado no material de Graça Pimentel, Maria Cristina e Leandro C. Cintra.

Divisão do arquivo

PARTE 1

- Operações Cosequenciais
- Modelo para implementação de processos cosequenciais
  - Intersecção de duas listas (*matching* de nomes em duas listas)
  - Intercalação (*merging*) de duas listas em uma
  - Intercalação em K-vias

Divisão do arquivo

PARTE 2

- Revendo a ordenação em memória
- Sobreposição de Processamento e E/S: *Heapsort*
- Construção da *heap* simultaneamente à leitura do arquivo
- Ordenação Simultânea com Escrita

Divisão do arquivo

PARTE 3

- Intercalação como Método para Ordenação de Arquivos Grandes em Disco
- Quanto tempo custa o *MergeSort*?
  - Leitura dos registros para criação de corridas;
  - Escrita das corridas ordenadas para o disco;
  - Leitura das corridas do disco para a memória (para intercalação);
  - Escrita do arquivo final em disco;
  - Tempo total;
  - Comparação.
- Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

Divisão do arquivo

PARTE 4

- O custo de aumentar o tamanho do arquivo
- Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)
- Aumento do tamanho das corridas utilizando *Replacement Selection*
- *Replacement Selection* + Intercalação em Múltiplos Passos

Divisão do arquivo

PARTE 5

- Ordenação de arquivos em fita
  - Intercalação Balanceada
  - Intercalação Balanceada em K-vias (K-way Balanced Merge)
  - Intercalações em Múltiplas Fases
  - Ordenação Externa em Disco e em Fita
- Pacotes para Mergesort

Divisão do arquivo

PARTE 1

- Operações Cosequenciais
- Modelo para implementação de processos cosequenciais
  - Intersecção de duas listas (*matching* de nomes em duas listas)
  - Intercalação (*merging*) de duas listas em uma
  - Intercalação em K-vias

Operações Cosequenciais

- Consistem em operações que envolvem o processamento coordenado de duas ou mais listas de entrada sequenciais, de modo a produzir uma única lista como saída.
  - Esse tipo de operação é muito comum em arquivos.

- Exemplos?
  - *Merging* (concatenação, ou união);
  - *Matching* (intersecção) de duas ou mais listas;
  - A operação de união, por exemplo, é a base do processo de ordenação de arquivos muito grandes, cujo índice não cabe na memória principal.

Modelo para implementação de processos cosequenciais

- Inicialmente, será apresentado um modelo genérico para a construção de procedimentos cosequenciais simples, curtos e robustos.
- Apesar de serem conceitualmente simples, operações que envolvem processamento cosequencial são, muitas vezes, implementadas de forma confusa e mal organizada.
- O objetivo desse capítulo é introduzir um modelo genérico a partir do qual operações que envolvem combinações de dois ou mais arquivos de entrada possam ser implementadas.

Intersecção de duas listas (*matching* de nomes em duas listas)

- Dadas duas listas de nomes de pessoas, queremos produzir uma lista com os nomes comuns a ambas, assumindo que cada uma das listas originais não contém nomes repetidos e que estão ordenadas em ordem crescente.

Intersecção de duas listas (*matching* de nomes em duas listas)

Lista 1	Lista 2
Adams	Adams
Carter	Anderson
Chin	Andrews
Davis	Bech
Foster	Rosewald
Garwich	Schmidt
Rosewald	Thayer
Turner	Walker
	Willis

Intersecção de duas listas (*matching* de nomes em duas listas)

- A idéia básica a ser seguida pelo algoritmo é a seguinte:

  Lê um nome de cada lista e os compara. O que fazer a seguir, depende do resultado dessa comparação...

  1. Se ambos são iguais, copiar o nome para a saída e avançar para o próximo nome em cada arquivo.
  2. Se o nome da Lista1 é menor, copiá-lo para a saída e avançar na Lista1 (lendo o próximo nome).
  3. Se o nome da Lista1 é maior, copiar o nome da Lista2 para a saída e avançar na Lista2.

Intersecção de duas listas (*matching* de nomes em duas listas)

- Pontos importantes a serem considerados pelo algoritmo:
  - **inicialização** (como abrir os arquivos e inicializar as informações para o processo funcionar corretamente);
  - **sincronização** (como avançar adequadamente em cada arquivo);
  - **gerenciamento de condições de fim-de-arquivo** (o processo deve parar ao atingir o fim de uma das listas);
  - **reconhecimento de erros** (nomes duplicados ou fora de ordem).

Intersecção de duas listas (*matching* de nomes em duas listas)

- Sincronização
  A cada passo do processamento das duas listas, um nome corrente da Lista1 e outro da Lista2. Sejam Nome1 e Nome2 as variáveis que armazenam esses dados. O algoritmo deve comparar Nome1 (da Lista1) e Nome2 (da Lista2):
  - se Nome1 é menor que Nome2, lemos o próximo nome da Lista1;
  - se Nome1 é maior que Nome2 lemos o próximo nome da Lista2;
  - se são iguais, escrevemos o nome na lista de saída, e lemos os próximos nomes de ambos os arquivos.

Intersecção de duas listas (*matching* de nomes em duas listas)

- O procedimento cosequencial para intersecção é baseado num único loop, que controla a continuação do processo através de um *flag*. Dentro do loop tem-se um comando condicional triplo para testar cada uma das condições anteriores, e o controle volta ao *loop* principal a cada passo da operação.
- O corpo do procedimento principal não se preocupa com EOF ou com erros na sequência de nomes. Para garantir uma lógica clara ao procedimento principal, esta tarefa foi deixada para a função *Input*(), que verifica a ocorrência de fim de arquivo, bem como a ocorrência de nomes duplicados ou fora de ordem...

Intersecção de duas listas (*matching* de nomes em duas listas)

- Finalmente, o procedimento *Initialize*( ):
  - abre arquivos de entrada e de saída;
  - seta o *flag* MORE_NAMES_EXIST como TRUE;
  - inicializa as variáveis que armazenam o nome anterior (uma para cada lista, denominadas PREV_1 e PREV_2) para um valor que garantidamente é menor que qualquer valor de entrada (LOW_VALUE). Isso garante que a rotina *Input*( ) não precisa tratar a leitura dos 2 primeiros registros como um caso especial.

Intersecção de duas listas (*matching* de nomes em duas listas)

MERGE

```
call initialize()
call input() to get NAME_1 from LIST_1 and NAME_2  from LIST_2
while(MORE_NAMES_EXIST)
    if (NAME_1 < NAME_2)
        write NAME_1 to OUT_FILE
        call input() to get NAME_1 from LIST_1
    else if (NAME_1 > NAME_2)
        write NAME_2 to OUT_FILE
        call input() to get NAME_2 from LIST_2
    else /* match – names are the same */
        write NAME_1  to OUT_FILE
        call input to get NAME_1 from LIST_1
        call input to get NAME_2 from LIST_2
finish_up()
```

Intersecção de duas listas (*matching* de nomes em duas listas)

Input()

- Argumentos de entrada
  - INP_FILE: descritor do arquivo de entrada (LIST_1 ou LIST_2)
  - PREVIOUS_NAME: nome lido da lista no passo anterior
  - OTHER_LIST_NAME: último nome lido da outra lista
- Argumentos de saída
  - NAME – nome lido
  - MORE_NAMES_EXIST - *flag* para indicar a parada

Intersecção de duas listas (*matching* de nomes em duas listas)

Input()

```
read next NAME from INP_FILE
If (EOF) and (OTHER_LIST_NAME == HIGH_VALUE)
   MORE_NAME_EXIST := FALSE /* fim das duas listas
else if (EOF)
   NAME:=HIGH_VALUE /* essa lista acabou
else if (NAME <= PREVIOUS_NAME) /* erro seqüencia
   Msg. de erro, aborta processamento
endif
PREVIOUS_NAME:=NAME
```

Intercalação (*merging*) de duas listas em uma

- O modelo anterior pode ser facilmente estendido para fazer a intercalação (união) de 2 listas, gerando como saída uma única lista ordenada (sem repetições de nomes).
- Neste caso, dois nomes, um de cada lista são comparados, e um nome é gerado na saída a CADA passo do comando condicional.
- O algoritmo compara os nomes e manda para a saída o dado menor, avançando no arquivo de onde saiu esse dado. Além disso, o processamento continua enquanto houver nomes em uma das listas. Ambos os arquivos de entrada devem ser lidos até o fim.

Intercalação (*merging*) de duas listas em uma

- Mudanças na rotina *Input*( ):
  - Mantém a *flag* MORE_NAMES_EXIST em TRUE enquanto existirem entradas em qualquer um dos arquivos, mas...
  - Não fica lendo do arquivo que já terminou:
    - solução: seta NAME (1 ou 2, dependendo do arquivo que terminou antes) para um valor que não pode ocorrer como entrada válida, e que seja maior em valor que qualquer entrada válida (definida na constante HIGH_VALUE).
  - Usa também OTHER_LIST_NAME para a função saber se a outra lista já terminou...

Intercalação em K-vias

- Não há motivo para restringir o número de entradas na intercalação a 2.
- Podemos generalizar o processo para intercalar k corridas simultaneamente.
- *k-Way MergeSort*

Intercalação em K-vias

- Esta solução
  - pode ordenar arquivos realmente grandes;
  - geração das corridas envolve apenas acesso seqüencial aos arquivos;
  - a leitura das corridas e a escrita final também só envolve acesso seqüencial;
  - aplicável também a arquivos mantidos em fita, já que E/S é seqüencial.

Divisão do arquivo

PARTE 2

- Revendo a ordenação em memória
- Sobreposição de Processamento e E/S: *Heapsort*
- Construção da *heap* simultaneamente à leitura do arquivo
- Ordenação Simultânea com Escrita

Revendo a ordenação em memória

- Ordenação de um arquivo que cabe em memória.
- Passos:
  - Lê arquivo para memória;
  - Ordena usando um bom algoritmo de ordenação em memória (por exemplo, *Shellsort*);
  - Escreve arquivo de volta no disco.

Revendo a ordenação em memória

- O tempo total de ordenação é igual à soma dos tempos de execução de cada um dos passos. O desempenho é bom, pois usa E/S sequencial.
- No caso de arquivos que cabem na memória, como melhorar o desempenho se já estão sendo utilizados os melhores métodos para Leitura, Escrita e Ordenação Interna?
- O que fazemos para melhorar o desempenho de algoritmos que têm vários passos se não é possível melhorar o desempenho de cada passo? Podemos tentar realizar alguns passos em paralelo!

Sobreposição de Processamento e E/S: *Heapsort*

- Para ordenação do arquivo em memória: devemos esperar que todo o arquivo seja lido antes de começar a trabalhar.
- Mas existe algum algoritmo de ordenação em memória interna rápido que possa começar a trabalhar a medida que os números são lidos? Existe o *Heapsort*, que lembra a árvore de seleção vista... a idéia é comparar chaves a medida que elas são encontradas.
- Árvore de seleção: a cada passo escreve a menor chave para o arquivo de saída... mas isso só é possível porque todo o arquivo está em memória.

Sobreposição de Processamento e E/S: *Heapsort*

- *Heapsort*: mantém as chaves numa estrutura chamada heap, que é uma árvore binária com 3 propriedades:
  - cada nó possui uma única chave, que é menor ou igual a chave do nó pai,
  - é uma árvore binária completa, i.e., todas as folhas estão em, no máximo, dois níveis, e todas as folhas do nível mais baixo estão em subárvores esquerdas;
  - deste modo, a heap pode ser mantida em um vetor tal que os índices dos filhos esquerdo e direito de um nó estão nas posições 2i e 2i+1;
  - respectivamente; o nó pai de um nó j é o menor inteiro igual a j/2.
- A última propriedade garante que a *heap* é um vetor no qual as posições das chaves são suficientes para impor ordem ao conjunto de chaves.

Construção da *heap* simultaneamente à leitura do arquivo

- *Heapsort* (algoritmo em duas partes):
  - construção da *heap* (a partir da leitura das chaves);
  - escrita das chaves ordenadamente (ordenação).
- A primeira parte pode ocorrer (virtualmente) simultaneamente à leitura das chaves, o que torna o seu custo computacional igual a zero: sai de graça em termos de tempo de execução.
- A implementação em UNIX desse tipo de estratégia é relativamente simples.

Construção da *heap* simultaneamente à leitura do arquivo

- Passos básicos da parte 1:

  for i := 1 to RECORD_COUNT

  read in the next record and appends it to the end of the array, call its key K

  while K is less than the key of its parent:

  exchange the record with key K and its parent

  next i

- Dados os passos do algoritmo, como realizar leitura e *heaping* simultaneamente?

Ordenação Simultânea com Escrita

- *Heapsort*:
  - construção da *heap;*
  - escrita das chaves ordenadamente.

Ordenação Simultânea com Escrita

- Terminada primeira parte, a segunda consiste em escrever a *heap* ordenadamente, cujo algoritmo é:

for i:= 1 to RECORD_COUNT

output the record in the first position in the array (*a menor chave!*)

move the key in the last position in the array (call it K) to the first position, and define the heap as having one fewer member that it previously had.

while K is larger than both of its children: exchange K with the smaller of its two children's keys

next i

Ordenação Simultânea com Escrita

- Não é imediata a possibilidade de realizar processamento e escrita simultaneamente. Mas sabemos de imediato qual será o primeiro registro do arquivo ordenado; e, depois de algum processamento, sabemos quem será o segundo, o terceiro, etc...
- Então, logo que um bloco de registros se encontra ordenado, ele pode ser escrito para o arquivo de saída, enquanto o próximo bloco é processado. E assim por diante.

Ordenação Simultânea com Escrita

- Além disso, cada bloco liberado para escrita pode ser considerado um novo *buffer* de escrita. É necessário uma coordenação cuidadosa entre processamento e escrita, mas, teoricamente pelo menos, as operações podem ser quase que completamente simultâneas.
- Vale notar que, nesse algoritmo, toda operação de E/S é essencialmente realizada sequencialmente: todos os registros são lidos segundo a ordem de ocorrência no arquivo, e escritos ordenadamente.

Ordenação Simultânea com Escrita

- Portanto, essa técnica funcionaria igualmente bem no caso de arquivos mantidos em disco ou fita. Além disso, sabemos que o menor número possível de operações de *seek* é realizado, pois toda E/S é sequencial.

Divisão do arquivo

PARTE 3

- Intercalação como Método para Ordenação de Arquivos Grandes em Disco
- Quanto tempo custa o *MergeSort*?
  - Leitura dos registros para criação de corridas;
  - Escrita das corridas ordenadas para o disco;
  - Leitura das corridas do disco para a memória (para intercalação);
  - Escrita do arquivo final em disco;
  - Tempo total;
  - Comparação.
- Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

Intercalação como Método para Ordenação de Arquivos Grandes em Disco

- Vimos o algoritmo *keysort*, que dava uma solução para ordenar um arquivo grande demais para caber na MEMÓRIA:
  - as chaves tinham que caber na memória, portanto arquivos realmente grandes não podem ser ordenados usando *keysort*,
  - após a ordenação das chaves é necessário realizar muitos *seeking*s para criar o arquivo ordenado,

Intercalação como Método para Ordenação de Arquivos Grandes em Disco

- Suponhamos um arquivo com 800.000 registros; cada registro com 100 bytes; cada registro contém uma chave com 10 bytes. Tamanho total do arquivo: 80 MB.
- Suponhamos também que temos 1 MB de memória de trabalho (sem contar o programa, sistema operacional, *buffers* de E/S etc).
- Nesse caso, é impossível ordenar em MEMÓRIA, mesmo usando o *keysort*.

Intercalação: Método para Ordenação de Arquivos Grandes em Disco

- A intercalação em k-vias (*k-way merging*) sugere uma solução. Uma vez que algoritmos em MEMÓRIA, como o *heapsort*, podem ser utilizados para trabalhar tanto em MEMÓRIA como em disco ou fita ao custo de um pequeno *overhead*, 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)

Intercalação: Método para Ordenação de Arquivos Grandes em Disco

- Cada sub-arquivo ordenado é chamado uma **corrida** (*run*). No exemplo anterior, cada corrida poderia conter 10.000 registros. (1 milhão de bytes, divididos por 100 bytes por registro).
- Uma vez criada a primeira corrida, pode-se repetir 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 uma intercalação em 80-vias para criar um arquivo completamente ordenado.

Intercalação: Método para Ordenação de Arquivos Grandes em Disco

- Esta solução:
  - pode ordenar arquivos realmente grandes;
  - a geração das corridas utiliza leitura sequencial (muito mais rápido que o *keysort*);
  - as leituras das corridas e a escrita final também são sequenciais.
  - se o *heapsort* é utilizado na parte da intercalação realizada em MEMÓRIA, a intercalação pode ser realizada simultaneamente a E/S, e a parte de intercalação em MEMÓRIA não aumenta o tempo total de execução;
  - fitas podem ser utilizadas, pois a E/S é, em sua maior parte, sequencial.

Quanto tempo custa o *MergeSort*?

- Supondo
  - Arquivo com 80 MB, e cada corrida com 1 MB.
  - 1MB = 10.000 registros.
  - Arquivo armazenado em áreas contíguas do disco (extents), extents alocados em mais de uma trilha, de tal modo que um único rotational delay é necessário para cada acesso.
  - Características do disco
    - tempo médio para seek: 18 ms
    - atraso rotacional: 8.3 ms
    - taxa de transferência: 1229 bytes/ms
    - tamanho da trilha: 20.000 bytes

Quanto tempo custa o *MergeSort*?

- 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 sequencial.
  - *extent*s alocados em mais que uma trilha são alocados de tal modo que apenas um *rotational delay* é necessário para cada acesso.

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.
    4 - escrita do arquivo final em disco.

Quanto tempo custa o *MergeSort*?

1 – leitura dos registros para a memória para a criação de corridas

- Lê-se 1MB de cada vez, para produzir corridas de 1 MB.
- Serão 80 leituras, para formar as 80 corridas iniciais.
- O tempo de leitura de cada corrida inclui o tempo de acesso a cada bloco (seek + rotational delay) somado ao tempo necessário para transferir cada bloco.

Quanto tempo custa o *MergeSort*?

1 – leitura dos registros para a memória para a criação de corridas

seek = 18ms, *rot. delay* = 8.3ms, total 26.3ms

Tempo total para a fase de ordenação:

80*(tempo de acesso a uma corrida) + tempo de transferência de 80MB

Acesso: 80*(*seek*+*rot.delay*= 26.3ms) = 2s

Transferência: 80 MB a 1.229 bytes/ms = 65s

Total: 67s

Quanto tempo custa o *MergeSort*?

2 – Escrita das corridas ordenadas para o disco

- Idem à leitura!

✓ Serão necessários outros 67s

Quanto tempo custa o *MergeSort*?

3 – Leitura das corridas do disco para a memória (para intercalação

- 1MB de MEMÓRIA para armazenar 80 *buffer*s de entrada
  - Portanto, cada *buffer* armazena 1/80 de uma corrida (12.500 bytes). Logo, cada corrida deve ser acessada 80 vezes para ser lida por completo.
- 80 acessos para cada corrida X 80 corridas
  - 6.400 *seek*s
- considerando acesso = *seek* + *rot. delay*
  - 26.3ms X 6.400 = 168s
- Tempo para transferir 80 MB = 65s

Quanto tempo custa o *MergeSort*?

4 – Escrita do arquivo final em disco

- Precisamos saber o tamanho dos *buffer*s 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, assumimos que é possível alocar 2 *buffer*s adicionais de 20.000 bytes para escrita
  - dois para permitir *double buffering*, 20.000 porque é o tamanho da trilha no nosso disco hipotético.

Quanto tempo custa o *MergeSort*?

4 – Escrita do arquivo final em disco

- Com *buffer*s de 20.000 bytes, precisaremos de 80.000.000 bytes / 20.000 bytes = 4.000 *seek*s.
- Como tempo de *seek*+*rot.delay* = 23.6ms por *seek*, 4.000 *seek*s usam 4.000 X 26.3, e o total de 105s.
- Tempo de transferência é 65s.

Quanto tempo custa o *MergeSort*?

Tempo total

- Leitura dos registros para a memória para a criação de corridas: 67s
- Escrita das corridas ordenadas para o disco: 67s
- Leitura das corridas para intercalação: 168 + 65 = 233 s
- Escrita do arquivo final em disco: 105 + 65 = 170 s
- Tempo total do *Mergesort* = 537 s

Quanto tempo custa o *MergeSort*?

Comparação

- Quanto tempo levaria um método que não usa intercalação?
  - Se for necessário um *seek* separado para cada registro, *i.e.*, 800.000 *seek*s a 26.3ms cada, o resultado seria um tempo total (só para *seek*) = 21.040s = 5 horas e 40s !

Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

- Antes de analisar o efeito de utilizarmos arquivos maiores, vamos ver os tipos de E/S realizados nas fases de ordenação e intercalação.

Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

- **Ordenação**
  - Acesso sequencial garantido, porque usamos *heapsort*. Como acesso sequencial implica em *seeking* mínimo, não é possível melhorar o tempo de I/O através de algoritmos: isso porque os registros têm que ser lidos e escritos pelo menos uma vez.

Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

- **Intercalação - leitura das corridas: acesso aleatório**.
  - Existe um *buffer* em MEMÓRIA para cada corrida, que são utilizados para I/O usando acesso aleatório.
  - O número de *buffer*s em MEMÓRIA para leitura dos dados em cada corrida determina o número de acessos aleatórios a serem realizados. Se esses *buffer*s puderem ser reconfigurados de modo a reduzir o número de acessos aleatórios, as operações de I/O podem ser melhoradas correspondentemente.
  - Portanto, nossa melhor chance é procurar métodos para diminuir os número de acessos aleatórios que ocorrem quando as corridas são lidas na fase de intercalação.

Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

- **Intercalação - escrita do arquivo final - também sequencial**.

Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

- **Análise** - arquivo de 800 MB
  - 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 *buffer*s na fase de intercalação.

Ordenação de um arquivo que é 10 vezes maior (8.000.000 de registros)

- Neste caso, cada *buffer* comporta 1/800th de uma corrida, e cada corrida é acessada 800 vezes (ou seja, são necessários 800 *seek*s por corrida).
  - 800 corridas X 800 *seek*s/corrida = 640.000 *seek*s no total.
  - O tempo total agora é superior a 5 horas e 19 minutos, 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.

Divisão do arquivo

PARTE 4

- O custo de aumentar o tamanho do arquivo
- Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)
- Aumento do tamanho das corridas utilizando *Replacement Selection*
- *Replacement Selection* + Intercalação em Múltiplos Passos

O custo de aumentar o tamanho do arquivo

- A grande diferença de tempo na intercalação dos dois arquivos (de 80 e 800 MB) é conseqüência da diferença nos tempos de acesso às corridas (*seek* e *rotational delay*)
- Em geral, para uma intercalação em K-vias de K corridas, em que cada corrida é do tamanho da MEMÓRIA disponível, o tamanho do *buffer*s para cada uma das corridas é de:
  - (1/K) x tamanho da MEMÓRIA = (1/K) x tamanho de cada corrida

O custo de aumentar o tamanho do arquivo

- Como temos K corridas, a operação de intercalação requer K^2 *seek*s. Medido em termos de *seek*s, o *mergesort* é $O(K^2)$. Como K é diretamente proporcional a N , o *mergesort* é $O(N^2)$ em termos de *seek*s.
- Deste modo, podemos esperar que, conforme o arquivo cresce, o tempo requerido para realizar a ordenação cresce rapidamente.

O custo de aumentar o tamanho do arquivo

- Maneiras de reduzir esse tempo:
  - Usar mais hardware (disk drives, MEMÓRIA, canais de I/O);
  - Realizar a intercalação em mais de um passo, o que reduz a ordem de cada intercalação e aumenta o tamanho do buffer para cada corrida;
  - Aumentar o tamanho das corridas iniciais ordenadas;
  - Achar meios de realizar I/O simultâneo.

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)

- A diferença nos custos de acesso a disco e a MEMÓRIA é essencial quando se fala em estrutura de arquivos.
- Se o problema de ordenação envolvesse apenas operações em MEMÓRIA, os custos seriam medidos em termos do número de comparações necessárias para completar uma intercalação.

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)

- A intercalação em K-vias tem as seguintes características, ignorados os custos de *seeking*:
  - Cada registro é lido e escrito apenas uma vez;
  - Se a árvore de seleção for utilizada, o número de comparações requeridas para a intercalação em K-vias de N registros é função de N log K;
  - Uma vez que K é diretamente proporcional a N, a intercalação é N log N. Bastante eficiente se fosse em MEMÓRIA.

Diminuição do Número de *Seek*s usando Intercalação em Múltiplos Passos (*Multistep Merging*)

- Mas o problema agora é a ordenação de um arquivo que não cabe na MEMÓRIA! Para essa tarefa, talvez seja possível ter ganhos ao sacrificar as vantagens da (eficiente) intercalação em K-vias em troca de economia nos tempos de acesso....

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)

- Um modo de reduzir o número de *seeks* é reduzir número de corridas, o que permite alocar a cada corrida um *buffer* maior em memória. Um modo de fazer isso é usar mais MEMÓRIA. O outro é utilizar intercalação em múltiplos passos.

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)

- Na intercalação em múltiplos passos
  - Ao invés 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.

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)

- No exemplo do arquivo com 800 MB tinhamos 800 corridas com 10.000 registros cada. Para esse arquivo, a intercalação múltipla poderia ser realizada em dois passos:
  - primeiro, a intercalação de 25 conjuntos de 32 corridas cada,
  - depois, a intercalação em 25-vias.
- Nesse caso, cada registro é lido duas vezes... mas o uso de *buffer*s maiores diminui o *seeking*. O caso de passo único visto anteriormente exige 640.000 *seeks* nos arquivos de entrada.

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)

Para a intercalação em 2 passos, temos:

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

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*Multistep Merging*)

Para a intercalação em 2 passos, temos:

- **Segundo passo:**
  - Cada uma das 25 corridas de 32 MB pode alocar 1/25 do *buffer*, portanto, cada *buffer* pode alocar 400 registros, ou seja, 1/800 corrida. Então, esse passo exige 800 *seeks* por corrida, num total de 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 um tostão com memória adicional!

Diminuição do Número de *Seeks* usando Intercalação em Múltiplos Passos (*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 escrito duas vezes: mais 40.000 *seeks* (assumindo 2 *buffers* com 20.000 posiçõers cada).
- Somando tudo isso, o tempo total de intercalação = 5.907s ~ 1hora 38 min.
  - A intercalação em 800 vias consumia ~5 horas…

Aumento do tamanho das corridas utilizando *Replacement Selection*

- Se pudéssemos alocar corridas com 20.000 registros, ao invés de 10.000 (limite imposto pelo tamanho da MEMÓRIA), teríamos uma intercalação em 400-vias, ao invés de 800.
- Neste caso, seriam necessários 800 *seek*s por corrida, e o número total de *seek*s seria: 800 *seek*s/corrida X 400 corridas = 320.000 *seek*s.
  - Portanto, dobrar o tamanho das corridas reduz o número de *seek*s pela metade.
- Como aumentar o tamanho das corridas iniciais sem usar mais memória?

Aumento do tamanho das corridas utilizando *Replacement Selection*

- Idéia básica: selecionar na memória a menor chave, escrever esse registro no *buffer* de saída, e usar seu lugar (*replace it*) para um novo registro (da lista de entrada).
- Os passos são:
  1. Leia um conjunto de registros e ordene-os utilizando *heapsort*, criando uma *heap* (*heap* primária).
  2. Ao invés de escrever, neste momento, a *heap* primária inteira ordenadamente e gerar uma corrida (como seria feito no *heapsort* normal), escreva apenas o registro com menor chave.

Aumento do tamanho das corridas utilizando *Replacement Selection*

3. Busque um novo registro no arquivo de entrada e compare sua chave com a chave que acabou de ser escrita

   Se ela for maior

   insira o registro normalmente na heap

   Se ela for menor que qualquer chave já escrita

   insira o registro numa heap secundária

4. Repita o passo 3 enquanto existirem registros a serem lidos. Quando a *heap* primária fica vazia, transforme a *heap* secundária em primária, e repita os passos 2 e 3.

Aumento do tamanho das corridas utilizando *Replacement Selection*

- **Perguntas:**
  - Dadas P posições de memória, qual o tamanho, em média, da corrida que o algoritmo de *replacement selection* vai produzir?
  - Quais são os custos de se usar r*eplacement selection*?
- **Respostas:**
  - A corrida terá, em média, tamanho 2P. (por quê?)
  - Precisaremos de buffers p/ I/O e, portanto, não poderemos utilizar toda a MEMÓRIA disponível para ordenação.

Replacement Selection + Intercalação em Múltiplos Passos

- Uma vez que a intercalação em 2 passos melhora em muito o desempenho, o replacement selection exatamente na forma como foi visto não seria usado...

Replacement Selection + Intercalação em Múltiplos Passos

- No caso dos exemplos acima, utilizando intercalação em 2 passos ao invés de intercalação simples teríamos:
  - ordenações em MEMÓRIA: 800 corridas, 25x32 *way* + 25 *way merge*, 127.000 *seek*s, 56 min
  - *Replacement selection*: 534 corridas, 19x28 *way* + 19 *way merge*, 124.438 *seek*s, 55 min
  - *Replacement selection*: 200 corridas, 20x10 *way* + 20 *way merge*, 110.400 *seek*s, 48 min

Replacement Selection + Intercalação em Múltiplos Passos

- Portanto o uso de intercalação em 2 passos melhora todos os casos. O método utilizado para formar as corridas não é tão importante quanto utilizar intercalações múltiplas!
- Obs: As diferenças estão exageradas, pois não consideram tempo de transmissão.

FIM