Processamento Coseqüencial

Leandro C. Cintra

M.C.F. de Oliveira

Fonte: Folk & Zoelick, File Structures

Operações Coseqüenciais

- Envolvem o processamento coordenado (simultâneo) de duas ou mais listas de entrada seqüenciais, de modo a produzir uma única lista como saída
- Exemplo: merging (intercalação) ou matching (intersecção) de duas ou mais listas mantidas em arquivo

Modelo para implementação de processos coseqüenciais

Lista1	**Lista2**
Adams	Adams
Carter	Anderson
Chin	Andrews
Davis	Bech
Foster	Rosewald
Garwich	Schmidt
Rosewald	Thayer
Turner	Walker
	Willis

Modelo para implementação de processos coseqüenciais

- Algoritmo de intercalação
  - lê um nome de cada lista, e compara
  - se ambos são iguais, copia o nome para a saída e avança para o próximo nome da lista em cada arquivo
  - se o nome da Lista1 é menor, ele é copiado para a saída, e avança na Lista1
  - se o nome da Lista1 é maior, copia o nome da Lista2 para a saída e avança na Lista2

PROGRAM: 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()
```


PROCEDURE 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

PROCEDURE 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
```


Ordenação por Intercalação de Arquivos em Disco

- 2-Way Mergesort, ou intercalação em duas vias, em disco
- Ao invés de considerar os registros individualmente, podemos considerar blocos de registros ordenados (corridas, ou *runs*), para minimizar os seeks...
- Método envolve 2 fases: geração das corridas, e intercalação

2-Way Mergesort

- Fase 1: Geração das Corridas
  - segmentos do arquivo (corridas) são ordenados em memória RAM, usando algum método eficiente de ordenação interna (p.ex., Quicksort), e gravados em disco
  - corridas vão sendo gravadas a medida em que são geradas

9

2-Way Mergesort

- Fase 2: Intercalação
  - As corridas gerados na fase anterior são intercaladas, duas a duas, formando corridas maiores que são gravadas em disco
  - O processo continua até que uma única corrida seja obtida, que consiste de todo o arquivo ordenado

10

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*

11

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 tb. a arquivos mantidos em fita, já que E/S é seqüencial

12

Qual o custo (tempo) do 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

Qual o custo (tempo) do MergeSort ?

- Quatro passos a serem considerados
  - leitura dos registros, do disco para a memória, para criar as corridas
  - escrita das corridas ordenadas para o disco
  - leitura das corridas para intercalação
  - escrita do arquivo final em disco

Leitura dos registros e criação das 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

Leitura dos registros e criação das 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

Escrita das corridas ordenadas no disco

- Idem à leitura!

Serão necessários outros 67s

17

Leitura das corridas do disco para a memória (para intercalação)

- 1MB de MEMÓRIA para armazenar 80 buffers 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 seeks
- considerando acesso = seek + rot. delay
  - 26.3ms X 6.400 = 168s
- Tempo para transferir 80 MB = 65s

18

Escrita do arquivo final em disco

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

19

Escrita do arquivo final em disco

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

20

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

21

Comparação

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

22

Ordenação de um arquivo com 8.000.000 de registros

- Análise - arquivo de 800 MB
- O arquivo aumenta, mas a memória não!
  - Em vez de 80 corridas iniciais, teremos 800
  - Portanto, seria 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

23

Ordenação de um arquivo com 8.000.000 de registros

- Cada buffer comporta 1/800 de uma corrida, e cada corrida é acessada 800 vezes
- 800 corridas X 800 seeks/corrida = 640.000 seeks no total
- O tempo total agora é superior a 5 horas e 19 minutos, aproximadamente 36 vezes maior do que o arquivo de 80 MB (que é 10 apenas vezes menor do que este)

24

Ordenação de um arquivo com 8.000.000 de registros

Definitivamente: necessário diminuir o tempo gasto obtendo dados na fase de intercalação

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 buffers para cada uma das corridas é de:
  (1/K) x tamanho da MEMÓRIA = (1/K) x tamanho de cada corrida

Complexidade do Mergesort

- Como temos K corridas, a operação de intercalação requer K^2 seeks
- Medido em termos de seeks, o Mergesort é $O(K^2)$
- Como K é diretamente proporcional à N , o Mergesort é $O(N^2)$, em termos de seeks

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
- realizar I/O simultâneo à intercalação

Redução do número de seeks: Intercalação em Múltiplos Passos (*Multistep Merging*)

- ao invés de intercalar todas as corridas simultaneamente, o grupo original é dividido em sub-grupos menores
- intercalação é feita para cada sub-grupo
- para cada sub-grupo, 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

Intercalação em Múltiplos Passos

- É claro que um número menor de seeks será feito no primeiro passo. E no segundo?
- O segundo passo exige não apenas seeking, mas também transferências nas leituras/escritas. Será que as vantagens superam os custos?
- 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, uma intercalação em 25-vias

Intercalação em Múltiplos Passos

- passo único visto anteriormente exige 640.000 seeks. Para a intercalação em 2 passos, temos, no passo 1:
  - 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

Intercalação em Múltiplos Passos

- No passo 2, cada uma das 25 corridas de 32 MB pode alocar 1/25 do buffer
  - portanto, cada buffer aloca 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

E o tempo total de intercalação?

- Nesse caso, cada registro é 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 de 20.000 bytes 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

- 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 seeks por corrida, e o número total de seeks seria: 800 seeks/corrida X 400 corridas = 320.000 seeks.
  - Portanto, dobrar o tamanho das corridas reduz o número de seeks pela metade
- Como aumentar o tamanho das corridas iniciais sem usar mais memória?

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

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

Replacement Selection + Intercalação em Múltiplos Passos

- Para melhorar a eficiência: o método utilizado para formar as corridas é menos relevante do que utilizar intercalações múltiplas!

37