Ordenação Externa (Merge Sort)

Referências: A.V.Aho, J.E.Hopcroft, J.D.Ullman, Data Structures and Algorithms, Cap. 11.

- Problema: ordenar dados organizados como arquivos, ou de forma mais geral, ordenar dados armazenados em memória secundária $\rightarrow$ *ordenação externa*.

- O "Merge sort" é capaz de ordenar um arquivo de n registros com $O(\log n)$ varreduras pelo arquivo

- Idéia central do algoritmo: organizamos um arquivo em *séries* progressivamente maiores, ou seja, seqüências de registros $r_1 \ldots r_k$ em ordem não decrescente ($r_i \leq r_{i+1}, 1 \leq i < k$).

- Dizemos que um arquivo está organizado em *séries* de comprimento k se seus registros formarem blocos

  $$r_1, r_2, \ldots, r_k | r_{k+1}, r_{k+2}, \ldots, r_{2k} | \ldots | r_{(i-1)k+1}, r_{(i-1)k+2}, \ldots, r_{ik} |$$
  $$\ldots | r_{m-q+1}, r_{m-q+2}, \ldots, r_m$$

  onde

  a) Para todo $i \geq 0$ tal que $ik \leq m$, $r_{(i-1)k+1} \leq r_{(i-1)k+2} \leq \ldots \leq r_{ik}$;

  b) Se m não for divisível por k e $m = pk + q$, onde $q < k$, então $r_{m-q+1} \leq r_{m-q+2} \leq \ldots \leq r_m$.
  Note que $r_{m-q+1}, r_{m-q+2} \ldots r_m$ (denominada *cauda*)é uma série de comprimento q.

- Exemplo: Arquivo com séries de comprimento 3:

7	15	29	8	11	13	16	22	31	5	12

- O passo básico do merge sort em arquivos é começar com dois arquivos, digamos f_1 e f_2, organizados em séries de comprimento k. Assuma que:

  1. o número de séries em f_1 e f_2, incluindo a cauda, difere de no máximo 1;
  2. no máximo um dentre f_1 e f_2 tem uma cauda; e
  3. o arquivo com a cauda tem pelo menos tantas séries quanto o outro.

- Com as premissas acima, é um processo simples ler uma série de cada arquivo f_1 e f_2, indercalar as séries e concatenar a série resultante (de comprimento $2k$) em um dentre dois arquivos g_1 e g_2, os quais estão sendo organizados em séries de comprimento $2k$.

- Alternando entre g_1 e g_2, podemos fazer com que esses arquivos não apenas estejam organizados em séries de comprimento $2k$, mas satisfaçam as propriedades (1), (2) e (3) acima.

- Para ver que (2) e (3) são satisfeitos , observe que a cauda de f_1 ou f_2 se intercala na última série criada (ou talvez seja a própria).

- **Passos do algoritmo:**

  – Começamos dividindo todos os n registros em dois arquivos f_1 e f_2 o mais uniformemente possível. Todo arquivo pode ser visto como organizado em séries de comprimento 1.

  – Podemos intercalar então as séries de comprimento 1 e distribuí-las sobre os arquivos g_1 e g_2, organizados em séries de comprimento 2.

  – Esvaziamos f_1 e f_2, e intercalamos g_1 e g_2 sobre f_1 e f_2, que estarão então organizados em séries de comprimento 4.

  – Então intercalamos f_1 e f_2 para criar g_1 e g_2, organizados em séries de comprimento 8, e assim por diante.

28	3	93	10	54	65	30	90	10	69	8	22
31	5	96	40	85	9	39	13	8	77	10	

(a) initial files

28	31	93	96	54	85	30	39	8	10	8	10
3	5	10	40	9	65	13	90	69	77	22	

(b) organized into runs of length 2

3	5	28	31	9	54	65	85	8	10	69	77
10	40	93	96	13	30	39	90	8	10	22	

(c) organized into runs of length 4

3	5	10	28	31	40	93	96	8	8	10	10	22	69	77
9	13	30	39	54	65	85	90							

(d) organized into runs of length 8

3	5	9	10	13	28	30	31	39	40	54	65	85	90	93	96
8	8	10	10	22	69	77									

(e) organized into runs of length 16

3 5 8 8 9 10 10 10 13 22 28 30 31 39 40 54 65 69 77 85 90 93 96

(f) organized into runs of length 32

```
procedure merge ( k: integer;  { the input run length }
        f1, f2, g1, g2: file of recordtype );
var
    outswitch: boolean;  { tells if writing g1 (true) or g2 (false) }
    winner: integer;  { selects file with smaller key in current record }
    used: array [1..2] of integer;  { used[j] tells how many
        records have been read so far from the current run of file f_j }
    fin: array [1..2] of boolean;  { fin[j] is true if we have
        finished the run from f_j – either we have read k records,
        or reached the end of the file of f_j }
    current: array [1..2] of recordtype;  { the current records
        from the two files }


  procedure getrecord ( i: integer );  { advance file f_i, but
          not beyond the end of the file or the end of the run.
          Set fin[i] if end of file or run found }
      begin
          used[i] := used[i] + 1;
          if (used[i] = k) or
              (i = 1) and eof (f 1) or
              (i = 2) and eof (f 2) then fin[i] := true
          else if i = 1 then read(f 1, current[1])
          else read(f 2, current[2])
      end;  { getrecord }
```

```
begin  { merge }
    outswitch := true;  { first merged run goes to g 1 }
    rewrite (g 1); rewrite (g 2);
    reset (f 1); reset (f 2);
    while not eof (f 1) or not eof (f 2) do begin  { merge two files }
        { initialize }
        used [1] := 0; used [2] := 0;
        fin [1] := false; fin [2] := false;
        getrecord (1); getrecord (2);
        while not fin [1] or not fin [2] do begin  { merge two runs }
            { select winner }
            if  fin [1] then winner  := 2
                  { f 2 wins by "default" – run from f 1 exhausted }
            else if fin [2] then winner  := 1
                { f 1 wins by default }
            else  { neither run exhausted }
                if current [1].key  < current [2].key  then winner  := 1
                else winner  := 2;
            { write winning record }
            if outswitch  then write (g 1, current [winner ])
            else write (g 2, current [winner ]);
            { advance winning file }
            getrecord (winner )
        end;
        { we have finished merging two runs - switch output
            file and repeat }
        outswitch := not outswitch
    end
end;  { merge }
```

- **Análise de complexidade**:
  - Após i passos, temos dois arquivos consistindo de séries de comprimento 2^i. Se $2^i \geq n$, então um dos dois arquivos estará vazio, e o outro conterá uma única série de comprimento n, ou seja, estará ordenado. Como $2^i \geq n \Leftrightarrow i \geq \log_2 n$, então $\lceil \log_2 n \rceil$ passos são suficientes.
  - Cada passo requer a leitura de dois arquivos e a escrita de dois arquivos, cada um com comprimento aproximado $n/2$. O número total de blocos de disco lidos ou escritos em um passo é então aproximadamente $2n/b$, onde b é o número de registros que cabem em um bloco.
  - Portanto, o número total de leituras e escritas do processo inteiro de ordenação é $O((n \log_2 n)/b)$.

- **Agilizando o Merge Sort:**

  Ao invés de iniciar o processo com arquivos organizados em séries de comprimento 1, pode-se começar com um passo que, para algum k apropriado, lê cada grupo de k registros contíguos para a memória principal, ordena-os (via quicksort, por exemplo), e os escreve como uma série de comprimento k.

  Por exemplo, se tivermos $n = 10^6$ registros e começarmos com $k = 1$, precisaremos de 20 passos para o processo de ordenação.

  Se pudermos armazenar 10^4 registros na memória principal, podemos, em um passo, ler 100 grupos de comprimento 10^4, ordenar cada grupo, e assim iniciar com 100 séries de comprimento 10^4 distribuídos uniformemente entre dois arquivos.

  Com apenas 7 passos do algoritmo teremos o arquivo ordenado, uma vez que $10^4 \times 2^7 = 1.280.000$.

- **Merge Sort de vários caminhos (*multiway*):**
  - O algoritmo apresentado acima pode ser generalizado para m arquivos de leitura e m arquivos de escrita, ao invés de apenas 2. O raciocínio é análogo a 2 arquivos:
  - Dividimos os n registros em m arquivos $f_1, f_2, \ldots, f_m$, organizados em séries de tamanho k, obedecendo às restrições (1), (2), (3).

    Note que cada série é ordenada em memória principal.

    Por exemplo considere o arquivo abaixo, supondo $m = 3, k = 3$:

    Arquivo original:

I N T E R C A L A C A O B A L A N C E A D A

    Arquivos organizados em séries de tamanho 3:

f1	*I N T*	*A C O*	*A D E*
f2	*C E R*	*A B L*	*A*
f3	*A A L*	*A C N*	

  - Intercalamos então as séries de comprimento k dos arquivos $f_1 \ldots f_m$, distribuindo nos arquivos $g_1 \ldots g_m$ (que estarão organizados em séries de tamanho km.
  - Esvaziamos $f_1 \ldots f_m$, e intercalamos $g_1 \ldots g_m$ sobre $f_1 \ldots f_m$, que estarão então organizados em séries de comprimento km^2.
  - Repetimos o passo anterior (intercambiando entre os arquivos f e g para leitura/gravação) para obter m arquivos organizados em séries de tamanhos km^3, km^4, etc.

- **Complexidade do *multiway* Merge Sort:**
  - A intercalação na memória principal pode ser feita em tempo $O(\log_2 m)$ por registro, se usarmos um heap ou outra estrutura que permita as operações INSERT e DELETEMIN em tempo logarítmico.
  - Se tivermos n registros, e o comprimento das séries é multiplicado por m a cada passo, então após i passos as séries serão de tamanho km^i. Se $km^i \geq n$, ou seja, após $i = \log_m(n/k)$ passos, o arquivo inteiro estará ordenado.
  - Como $\log_m(n/k) = \log_2(n/k)/\log_2 m$, reduzimos por um fator de $\log_2 m$ o número de escritas e leituras de cada registro.
  - Além disso, se tivermos m unidades de disco para leitura de arquivos e m unidades para escrita, de tal forma que esses discos possam ser lidos/gravados em paralelo, os dados serão processados m vezes mais rápidos do que se tivéssemos uma unidade de disco para leitura e uma unidade para gravação; ou ainda $2m$ vezes mais rápido do que se tivéssemos apenas uma unidade de disco para escrita e gravação.
  - O incremento indefinido de m não aumenta a velocidade do processamento por um fator $log_2 m$: para um numero suficientemente grande de m, o tempo para fazer a intercalação em memória principal (custo $O(log_2 m)$) irá exceder o tempo de I/O em disco.

Exemplo: Tempo aproximado do Merge Sort $\times$ número de arquivos

numero de registros	n	1E+08
numero de chaves/bloco	b	1E+02
comprimento inicial das séries	k	1E+03
tempo por operação em memória (s)	TM	1E-06
tempo por r/w em disco (s)	TD	1E-03

Variável	Sigla / cálculo	Resultados							
quant. arquivos	m	2	4	8	16	32	64	128	256
passos	i	17	9	6	5	4	3	3	3
r/w por passo	RW = 2*n/b	2.0E+06	2.0E+06	2.0E+06	2.0E+06	2.0E+06	2.0E+06	2.0E+06	2.0E+06
r/w paralelo	RWP	1.0E+06	5.0E+05	2.5E+05	1.3E+05	6.3E+04	3.1E+04	1.6E+04	7.8E+03
r/w total	RWPT = RWP*i	1.7E+07	4.5E+06	1.5E+06	6.3E+05	2.5E+05	9.4E+04	4.7E+04	2.3E+04
tempo r/w total (s)	**TEMPRW = RWT*TD**	**1.7E+04**	**4.5E+03**	**1.5E+03**	**6.3E+02**	**2.5E+02**	**9.4E+01**	**4.7E+01**	**2.3E+01**
custo interc/reg	CIR = log2 (m)	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0
custo mem total	CM = CIR*i*n	1.7E+09	1.8E+09	1.8E+09	2.0E+09	2.0E+09	1.8E+09	2.1E+09	2.4E+09
tempo mem total (s)	**TEMPM = CM*TM**	**1.7E+03**	**1.8E+03**	**1.8E+03**	**2.0E+03**	**2.0E+03**	**1.8E+03**	**2.1E+03**	**2.4E+03**
tempo total (s)	**TTOTAL = TEMPM+TEMPRW**	**1.9E+04**	**6.3E+03**	**3.3E+03**	**2.6E+03**	**2.3E+03**	**1.9E+03**	**2.1E+03**	**2.4E+03**