EXTERNAL SORTING

Few Computer Engineers write sorting routines for large files. But many use & choose which packages to buy.

This chapter will help

- to estimate how much time sorting should take
- to make informed choices among sorting packages
- to decide under what circumstances to use the sorting package

4.1 Creating Initial Sorted Segments

Whenever a file does not fit entirely into memory:

- First Stage of Sorting : divide file into segments & sort each one separately.
- Second Stage of Sorting : merge sorted segments

Each stage involves reading & writing the file at least once:

- sorting w/one disk drive takes at least $4*T_x$
- sorting w/two disk drives takes at least $2*T_x$ (reading & writing time is overlapped)

CA Income Tax File

6 million records of 400 B each ≈ 2400 MB file.

10 MB of comp memory.

We need to list the records in order of a field for which there is no index and for which it is not already sorted.

Time for finding records in order, given a list of key values but no addresses of records:

= 80 years

Sorting would be a better option!

Heapsort with One Disk Drive

Most optimal method for external sorting with one disk drive

- Because input & output is overlapped, so only the I/O time needs to be counted. Sorting becomes free!

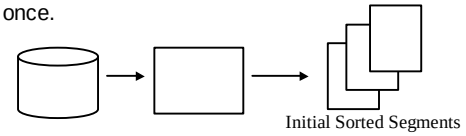
(This is not the case for other internal sorting routines such as quicksort, bubblesort, etc.)

Creates the initial sorted segments which are the size of the available memory.

Replacement Selection with Two Disk Drives

An extension of heapsort

- Initial sorted segments are twice the size of available memory
- Reading in unsorted file and writing out sorted segments overlap
- Initial stage takes the time to write the file once.



Merging Algorithms

- For 100 sorted segments, we take a piece from each segment & read it into memory.
  - When a piece of one sorted segment runs out, replace it with next piece of the same sorted segment.
- Do a 100-way merge in memory

Pros and Cons:

– requires a large # of disk accesses for pieces of segments.

+ it takes less time than using larger pieces & making several passes.

Overlapping Heapsort with I/O

Heap has two meanings:
- a pile file, or unsorted file
- priority queue [*]

A **priority queue** is a complete binary tree, where each node contains a record with a key value which is smaller than the keys in its two children.
- Root is the smallest key of all the records in the tree
- But keys are not totally in order

Complete Binary Trees

A **complete binary tree** has a tree in which
- all leaves are on two levels: k^{th} level and $(k+1)^{th}$ level,
- the leaves on the bottommost level are in the leftmost positions in that level.

A complete binary tree is easy to model with an array:
- We arrange the nodes of the tree in level order.
- The height is floor($\log n$), where n is the # nodes in the tree.
- The children of the i^{th} element in the array are in the $2i^{th}$ & $(2i+1)^{st}$ elements of the array. **See Figure 4.1**

The Idea of Heapsort

1st Stage
- read in records & place each new record
  - at the end of the array (in the rightmost bottom child position of the complete binary tree)
  - If new record is smaller than its parent's key, then exchange new record with its parent (recursively up the heap).
    (# comparisons & exchanges <= $\log n$, where n is the # records that have already been read into memory)
- We read in records until memory fills up.

2nd Stage
- The second stage overlaps writing out the sorted segment

See Figure 4.2

Example

400 B records in 2400 B blocks,10MB memory
- 6 insertions while reading the next block
- 25,000 records in the tree

Each record insertion causes at most
log 25000 = 14 exchanges
Each block has 6 blocks, so in worst case
about 84 comparisons & exchanges.
If each execution of exchange step is 10 μsec:
840 msec = 0.84 msec required
= time to read in one block of 2400 bytes in IBM 8330.
Assumptions:
- Some overlapping is possible
- Whole memory is used as a buffer
- It does not matter if T_I(heap) > btt (for next block)
- Most of heap insertion can overlap with reading

Output Sorted Segment

WHILE more records exist not written to disk **DO** **(Page 100)**
- Move the root record into the output buffer.
  Put the rightmost bottom record (say "X") in the root position.
- [heap condition test]
  If the value of X is greater than its two children, exchange it with the smaller child. Repeat this step for X, until the tree becomes a heap.

This is an O($\log n$) operation, where n is the # records left in the tree.
- It can be a great extent to be overlapped with the writing of the sorted segment.
- Seek time = 24.3 msec, T_R(for 30MB of data to be sorted) = 3500msec
- Time for creating the initial sorted segments = 2b*ebt
  - Time to read in segments overlapping creation of the heap and
  - time to write out the sorted segments, overlapping the 2nd phase of heapsort **(See Fig. 4.2)**

Ex: Cambridge Tax File

30,000 100-byte records can fit into 10 MB memory
Since heapsort overlaps I/O,
T_X = 1250*0.84 msec = 1 sec.
T_{sort} = 2b * ebt = 2 T_X = 2 sec
(1 sec to read in file & create a heap)
(1 sec to write out the sorted file)

(Compare with 4.38 hours required for ordered reading of records as computed in Chp.3)m

Replacement Selection w/2 Disks

- Produces sorted segments twice the size of memory.
- It takes only one sequential reading time.
  - (Reading, writing, and sorting all overlap.)

Algorithm:

- A heap is constructed from the first read segment.
- Using heapsort, smallest records that completes a block are emitted.
- A new block is read in:
  - New records with keys smaller than those of records already written out is put into a second heap in memory.
  - Others are inserted into the first heap as usual and smallest record at root position is moved to the output buffer.
  - The empty position (root position) in the first heap are filled with the smallest element of the second heap. (Exchanges may be needed)

Analysis:

- We can overlap i/o with internal processing:
  - We can both insert a record and output a record in the time it previously took just to do output of one record.

Extreme Cases:

- In case the keys of incoming records are mostly smaller than the keys of the records which have already been emitted:
  - After some time, the new heap in memory may become larger than the old one.
  - In this case we may not be able to overlap I/O (both usual input and usual output must take place)
- In case records are partially sorted in ascending order:
  - It will take longer before the new heap dominates the old heap. The output segment will be very large.

For random input, output segments are twice the size of memory.
- if file is partially sorted in the right order, the segments will be longer.
- opposite order, shorter.

Summary:

– Requires having two disks.
  – Must be able to write out blocks of sorted segments at the same time as reading in blocks of the unsorted file. **(Otherwise?)**
  (overlap is only possible after the memory has been filled once.)

+ This reduces the time for initial sorting of segments by a factor of two : b x ebt
+ Sorted segments are twice as large as the memory capacity.

One Disk Drive Large-Memory Merging

Assume we have a disk similar to IBM 8330 that has room for two 2400 MB files (sorted or unsorted) but only one disk arm.

Memory size = 10 MB

Heapsort was used to obtain 240 sorted segments, each 10MB of size(Time required is 14x2=28 min)

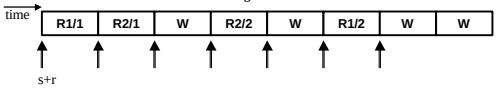
The Two-Way Merge

We merge two of the sorted segments at a time:

- We can read in half of each segment (5 MB) into memory
- One disk arm --> RW overlap or RR overlap is not possible!
- With double [output] buffering we can overlap merging w/ writing out.

Time for merging two sorted segments (each with ***seg*** blocks):

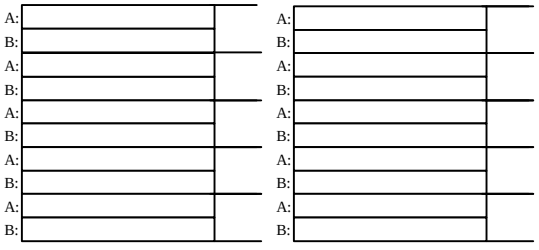
$\underbrace{(4 + 3) \times (r + s)}_{\text{\# half segments (R \& W)}} + 2 \times \underbrace{2 \times seg \times ebt}_{\text{R + W time for one segment}} = 170 + 14{,}400 = 14{,}170 \text{ ms} = 14.2 \text{ sec}$



(s+r) becomes significant in *n*-way merges, when *n* is large.

Example: Figure 4.5

A:	10	50	70	90	150	230	310	350	470	490
B:	20	60	100	200	250	300	400	520	610	700



A Two-Way Merge With 240 Segments

pass	1	2	3	4	5	6	7	8
segment size (MB)	10	20	40	80	160	7x320 1x160	3x640 1x480	1x1280 1x1120
number	240	120	60	30	15	8	4	2

We need ceiling($\log_2 240$) passes.

Time for first pass:

120 x 14.2 sec = 1704 sec = 28.4 min

For merging two 10MB sorted segments, we need (3 W +4 R)

For merging two 80MB sorted segments, we need (31 W+32 R)

After the first pass, (# R) @ (# W)

So, # separate seeks = 2 x #R = 2 x # pieces to be read in.

Time for one pass when we use a two-way merge :

$$2 \times \underbrace{2 \times (nsg)}_{\text{\# half segments}} \times (r + s) + \underbrace{2 \times b \times ebt}_{\text{R \& W whole file}}$$

(1 R & 1 W for each segment)

when we have *nsg* segments of memory size.

Back to our Example:

Time for each pass = ? 4 x 240 x (24.3) + 2,000,000x0.84=28.4 min

Time to do 8 passes = ? 8 x 28.4 min = 227 min

Sorting each segment originally = 28 min

Total = ? 255 min

The Four-way Merge

We merge four of the sorted segments at a time:

- We can read in **seg/4** of each segment (2.5 MB) into memory
- First pass : 60 segments of 40 MB each.
- Second pass: 15 segments of 160 MB
- Third Pass : 4 segments of . . .(3 x 640 and 1 x 480)
- Fourth Pass : 1 segment of 2400 MB.

Time for one four-way merge pass:

2 x 4 x (**nsg**) x (r + s) + 2 x b x ebt

Back to our example, what is the total time for sort + merge?

The *P*-way Merge

Time for one *P*-way merge pass:

2 x P x (**nsg**) x (r + s) + 2 x b x ebt

The number of passes: $\lceil \log_P(nsg) \rceil$

General Formula for Merge Sorting with One Disk Drive:

$$\underbrace{2b \times ebt}_{\text{sort time}} + \underbrace{\lceil \log_P(\mathbf{nsg}) \rceil}_{\text{\# passes}} \times [2 \times P \times (\mathbf{nsg}) \times (r + s) + 2 \times b \times ebt]$$

As a rule of thumb, try the following in order to decide what to choose for P:

P = **nsg**(one pass), (usually best for nsg < 196)

P = $\lceil \sqrt{(\mathbf{nsg})} \rceil$ (two passes),

and P = $\lceil \sqrt[3]{(\mathbf{nsg})} \rceil$ (three passes).

How the Merging is Done? (Figure 4.7)

In order to do the comparisons within the merge, we use another priority queue

- it consists of the lowest value record from each of the segments in memory.
- as a record is selected from the queue for the merge, the next lowest record from that segment is entered into the queue.

Sorting the Hospital File

Calculate the sort time for the hospital file with 100,000 of 400B patient records.

- Create 4 sorted segments using heapsort.
- Do one 4-way merge in memory.

Creating the Intersection File

Constitute the intersection file from MA & BCBS files, which are both 40 MB files. 70% of records are common in both files.

Two Disk Drive Large-Memory Merging

- We can do replacement selection
  - + twice the size of memory segments in average
  - + takes T_x time.
- We can overlap R & W in the merge phase.
  - + total sorting time is cut in half
  - - coordination of R and W in the merge is delicate.

Double Buffering the Input

We have a problem in overlapping R & W:

- Let's look at 2-way merging:
  - assume that two sorted segments of 20MB have roughly the same distribution. Read 5MB of data from each segment.
  - It is likely that both pieces will run out at about at the same time, and we shall be forced to wait because the other piece will be empty. We cannot R or W both segments simultaneously.

Double Buffering the Input

We have a problem in overlapping R & W:

- Let's look at 2-way merging:
  - assume that two sorted segments of 20MB have roughly the same distribution. Read 5MB of data from each segment.
  - It is likely that both pieces will run out at about at the same time, and we shall be forced to wait because the other piece will be empty. We cannot R or W both segments simultaneously.
  - One solution:
    - Use two buffers, each half the size we would use for one-disk drive merging, for each segment being merged (i.e., 2.5 MB)
    - This way when only half the records in memory from a given segment have been merged, new ones are read in from the same segment.
    - There might be occasional waiting periods but most of the time input time should overlap most of the output.

The CA Income Tax File

- See Slide #3.
- Initial sorting will produce 120 segments of about 20MB each (Takes 14 minutes)
- Let's perform a 120-way merge (one pass)
  - Each segment contributes 1/120 of the initial 10MB read into memory, i.e., 1/240 of the 20MB in each segment.
  - We use "half-size" buffers, so \$ 480 pieces in each segment to be read in separately (Each piece is slightly smaller than a track)
  - Assume that all data is kept in one disk, and all sorted data is written into another disk. We can overlap R and W.
  - Total time for merging=120 x 480 x (s + r) + b x *ebt* =2240 sec =37.3min

  (Compare this with 62 min that we computed for merging w/one disk drive)
- If we make 11-way merge (two passes)
  - Each segment is 44 pieces
  - Each pass takes 44 x 120 x 24.3 + 840,000 ms = 968 sec.
  - Two passes takes 1936 sec = 32.3 min, so total sort time=32+14=46min.

General Formula for Sorting with Two Disk Drives

$$b \times ebt + \lceil \log_P(\mathbf{nsg}) \rceil \times [4 \times P \times (\mathbf{nsg}) \times (r + s) + b \times ebt]$$

sort time; # passes

What is the formula for sorting with two disk drives, with a one-pass ***nsg***-way merge?

How More Disk Drives Could be Used?

If we have 3 or more disk drives, we could use

- one disk for W
- others for R and distribute sorted segments evenly on them

⇒ Seeks on one disk can overlap Reads on another disk

We could get close to optimal sorting time by managing to overlap R, seeking, merging, and W perfectly.

- This time can not be less than T_x.

The Intersection File

Sorting 40MB BCBS & MA files is very fast.

- Using replacement selection, we obtain 2 sorted segments of 20 MB each.
- We merge these two segments using a 2-way merge.

Total time $= b \times ebt + \lceil \log_P(nsg) \rceil \times [4 \times P \times (nsg) \times (r + s) + b \times ebt]$

$= 2 \times b \times ebt + 4 \times (nsg)^2 \times (r + s)$

$= 2 \times 16{,}667 \times 0{,}84 + 4 \times 4 \times 24.3 \equiv 28$ sec.

Seeks are not significant, and the sort time is near optimal ($2T_x$.)

Sorting both files takes 56 seconds.

We can overlap R & W, so writing intersection file takes 28 seconds.

So, the total time is 84 seconds.

(Compare this with 2.5 minutes it took with only one disk drive, and much faster than the 10.5 days it took without sorting.)

Using Quicksort with Virtual Memory is Slow !

- Quicksort is an O(*n* log *n*) sorting method
  - good if whole file fits into memory
    - as not good as heapsort, since quicksort cannot overlap I/O
  - if not, we can write a quicksort program that rely on virtual memory (i.e., as if the whole file fits into memory)
    - But this does not make sense if (file size) >> (memory size)
    - A paging policy such as LRU (least recently used) is often used in virtual memory implementation.
- Use quicksort in order to sort a file which is 240 times the size of memory (i.e., 2400 MB.)
- See Tharp (page 342-346)

Sorting Packages are Sometimes Very Slow !

- Sorting Package: a software that can be used to sort the given files using different algorithms.
- Package may only work well for files that fit in memory.
- Sorting in real situations may not work as well as the sorting under theoretical conditions:
  - multi-user environments
  - sub-optimal physical system configurations (not enough space on the disk for unsorted and sorted file)
  - not enough memory space is available for user's data

If a sorting package is 2 - 10 times slower than the ideal sorting described here, it is probably pretty good.