CMU SCS

Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 – DB Applications

Lecture 12: external sorting
(R&G ch. 13)

CMU SCS

Why Sort?

CMU SCS

Why Sort?

- select ... order by
  - e.g., find students in increasing *gpa* order
- *bulk loading* B+ tree index.
- *duplicate elimination* (select distinct)
- select ... group by
- *Sort-merge* join algorithm involves sorting.

CMU SCS

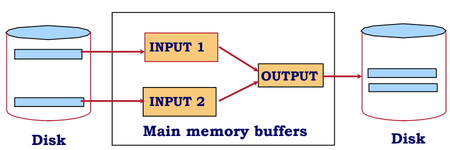
Outline

- ➡ two-way merge sort
- external merge sort
- fine-tunings
- B+ trees for sorting

CMU SCS

2-Way Sort: Requires 3 Buffers

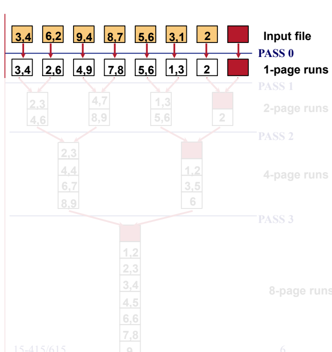
- Pass 0: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 1, 2, 3, …, etc.: requires 3 buffer pages
  - merge pairs of **runs** into runs twice as long
  - three buffer pages used.



CMU SCS

Two-Way External Merge Sort

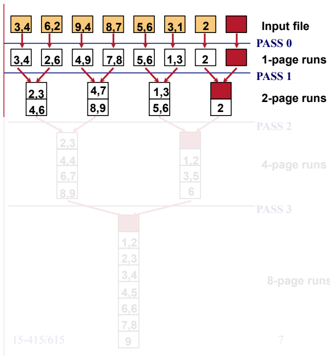
- Each pass we read + write each page in file.



CMU SCS

Two-Way External Merge Sort

- Each pass we read + write each page in file.

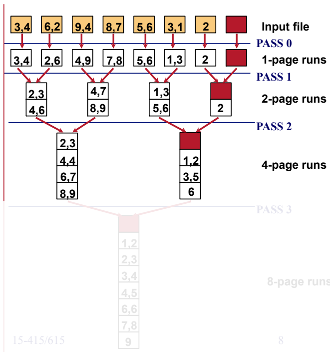


Faloutsos

CMU SCS

Two-Way External Merge Sort

- Each pass we read + write each page in file.

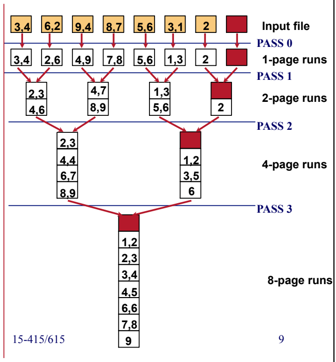


Faloutsos

CMU SCS

Two-Way External Merge Sort

- Each pass we read + write each page in file.



Faloutsos 15-415/615 9

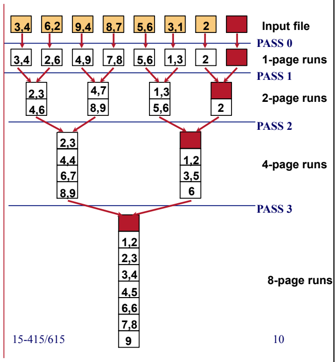
CMU SCS

Two-Way External Merge Sort

- Each pass we read + write each page in file.
- N pages in the file => $= \lceil \log_2 N \rceil + 1$
- So total cost is:

  $$2N\left(\lceil \log_2 N \rceil + 1\right)$$

- *Idea:* ***Divide and conquer:*** sort subfiles and merge



Faloutsos 15-415/615 10

CMU SCS

Outline

- two-way merge sort
- ➡ external merge sort
- fine-tunings
- B+ trees for sorting

Faloutsos 15-415/615 11

CMU SCS

External merge sort

B > 3 buffers

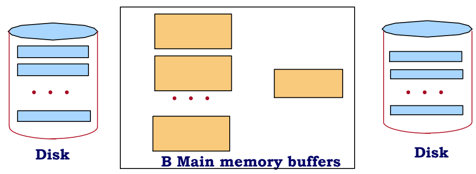
- Q1: how to sort?
- Q2: cost?

Faloutsos 15-415/615 12

CMU SCS

General External Merge Sort

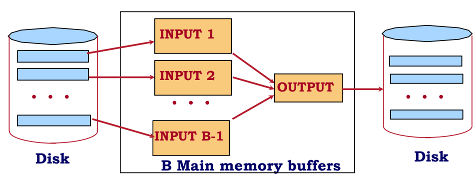
B>3 buffer pages. How to sort a file with N pages?



CMU SCS

General External Merge Sort

- Pass 0: use *B* buffer pages. Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
- Pass 1, 2, …, etc.: merge *B-1* runs.



CMU SCS

External merge sort

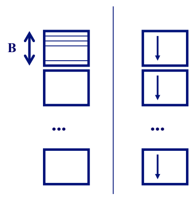
B > 3 buffers

- ✓ Q1: how to sort?
- Q2: cost?

CMU SCS

Sorting

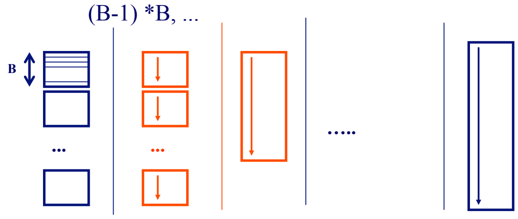
- create sorted runs of size B (how many?)
- merge them (how?)



CMU SCS

Sorting

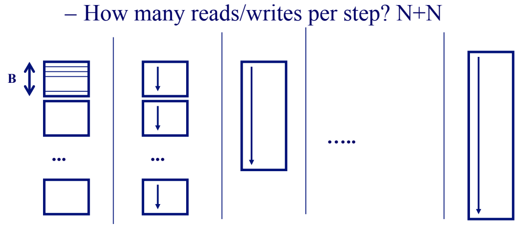
- create sorted runs of size B
- merge first B-1 runs into a sorted run of (B-1) *B, ...



CMU SCS

Sorting

- How many steps we need to do? 'i', where B*(B-1)^i > N
- How many reads/writes per step? N+N



CMU SCS

Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)

CMU SCS

Cost of External Merge Sort

- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Formula check: $\lceil \log_4 22 \rceil$ = 3 … + 1 → 4 passes ✓

CMU SCS

Number of Passes of External Sort

(I/O cost is 2N times number of passes)

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

CMU SCS

Outline

- two-way merge sort
- external merge sort
- ➡ fine-tunings
- B+ trees for sorting

CMU SCS

Outline

- two-way merge sort
- external merge sort
- fine-tunings
  - ➡ which internal sort for Phase 0?
  - blocked I/O
- B+ trees for sorting

CMU SCS

details

Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- But: we get B buffers, and produce 1 run of length B.
- Can we produce longer runs than that?

CMU SCS

details

Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- But: we get B buffers, and produce 1 run of length B.
- Can we produce longer runs than that?

Heapsort:
- Pick smallest
- Output
- Read from **next** buffer

CMU SCS

details

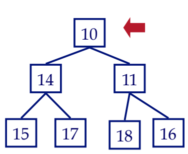
Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- But: we get B buffers, and produce 1 run of length B.
- Can we produce longer runs than that?
- Alternative: "tournament sort" (a.k.a. "heapsort", "replacement selection")
- Produces runs of length ~ 2*B
- Clever, but **not** implemented, for subtle reasons: tricky memory management on variable length records

CMU SCS

details

Reminder: Heapsort

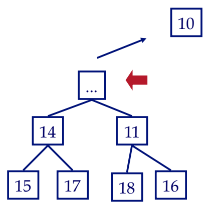


pick smallest, write to output buffer:

CMU SCS

details

Heapsort:



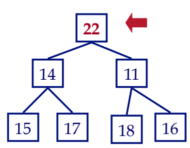
pick smallest, write to output buffer:

CMU SCS

details

Heapsort:

get next key; put at top and 'sink' it

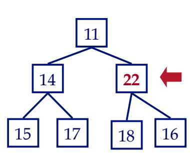


CMU SCS

details

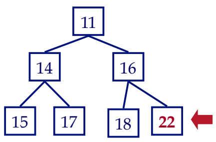
Heapsort:

get next key; put at top and 'sink' it



details

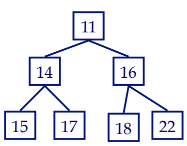
Heapsort:



get next key; put at top and 'sink' it

details

Heapsort:



When done, pick top (= smallest) and output it, if 'legal' (ie., >=10 in our example

This way, we can keep on reading new key values (beyond the B ones of quicksort)

Outline

- two-way merge sort
- external merge sort
- fine-tunings
  - which internal sort for Phase 0?
  - ➡ blocked I/O
- B+ trees for sorting

CMU SCS

Blocked I/O & double-buffering

- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?

CMU SCS

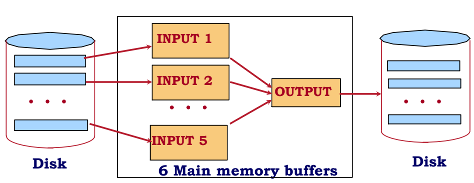
Blocked I/O & double-buffering

- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?
- A1: Blocked I/O (exchange a few r.d.a for several sequential ones) [use bigger pages]
- A2: double-buffering [mask I/O delays with prefetching]

CMU SCS

A1: blocked I/O

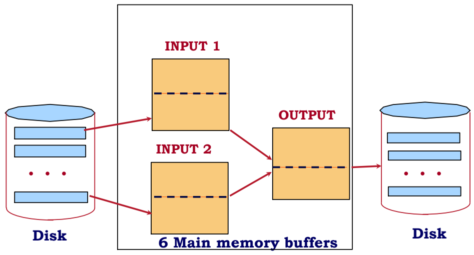
- Normally, ‘*B*’ buffers of size (say) 1K



CMU SCS

A1: blocked I/O

- Normally, '*B*' buffers of size (say) 1K
- INSTEAD: *B/b* buffers, of size '*b*' Kilobytes



CMU SCS

A1: blocked I/O

- Normally, '*B*' buffers of size (say) 1K
- INSTEAD: *B/b* buffers, of size '*b*' Kilobytes
- Pros?
- Cons?

CMU SCS

A1: blocked I/O

- Normally, '*B*' buffers of size (say) 1K
- INSTEAD: *B/b* buffers, of size '*b*' Kilobytes
- Pros? Fewer random d.a. (because some of them -> sequential)
- Cons? Smaller fanout -> maybe more passes

CMU SCS

Blocked I/O & double-buffering

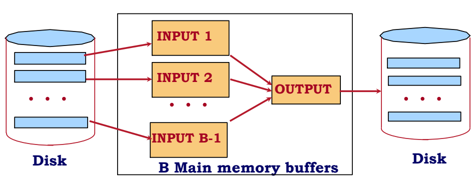
- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?
- A1: Blocked I/O (exchange a few r.d.a for several sequential ones) [use bigger pages]
- ➡ A2: double-buffering [mask I/O delays with prefetching]

Faloutsos 15-415/615 40

CMU SCS

A2: Double buffering

- Normally, when, say 'INPUT1' is exhausted
  - We issue a 'read' request and
  - We wait …

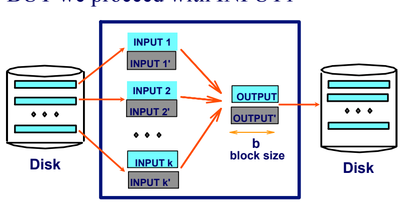


Faloutsos 15-415/615 41

CMU SCS

A2: Double Buffering

- w/ double bufferning, we *prefetch* INPUT1' into `*shadow block*'
  - When INPUT1 is exhausted, we issue a 'read',
  - BUT we proceed with INPUT1'

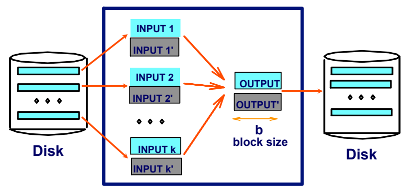


Faloutsos 42

CMU SCS

A2: Double Buffering

- Potentially, more passes; in practice, most files still sorted in 2-3 passes.



B main memory buffers, k-way merge

CMU SCS

Outline

- two-way merge sort
- external merge sort
- fine-tunings
- ➡ B+ trees for sorting

CMU SCS

Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- *Idea*: Can retrieve records in order by traversing leaf pages.
- *Is this a good idea?*
- Cases to consider:
  - B+ tree is clustered
  - B+ tree is not clustered

CMU SCS

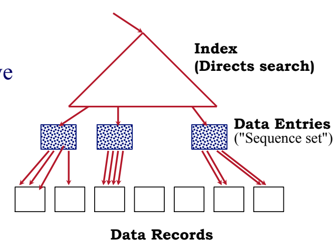
Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- *Idea*: Can retrieve records in order by traversing leaf pages.
- *Is this a good idea?*
- Cases to consider:
  - B+ tree is clustered Good idea!
  - B+ tree is not clustered Could be a very bad idea!

CMU SCS

Clustered B+ Tree Used for Sorting

- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)

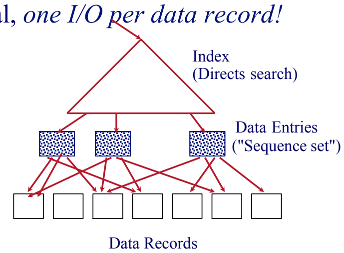


Always better than external sorting!

CMU SCS

Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, *one I/O per data record!*



External Sorting vs. Unclustered Index

N	Sorting	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

N: # pages
***p*: # of records per page**
B=1,000 and block size=32 for sorting
p=100 is the more realistic value.

Summary

- External sorting is important
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages).
  - Later passes: ***merge*** runs.
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.