External Sorting

Linda Wu

(CMPT 354 • 2004-2)

Topics

- Why sort
- Internal vs. external sorting
- Two-way external merge sort
- General external merger sort
- Replacement sort
- Blocked I/O
- Double buffering
- B+ tree for sorting

Why Sort?

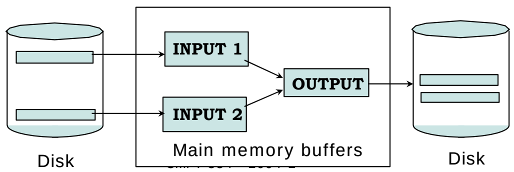
- A classic problem in computer science
- Data requested in sorted order
  - e.g., find students in increasing gpa order
- Sorting is first step in bulk loading B+ tree index
- Sorting is useful for eliminating duplicate copies in a collection of records
- Sort-merge join algorithm involves sorting

Internal vs. External Sorting

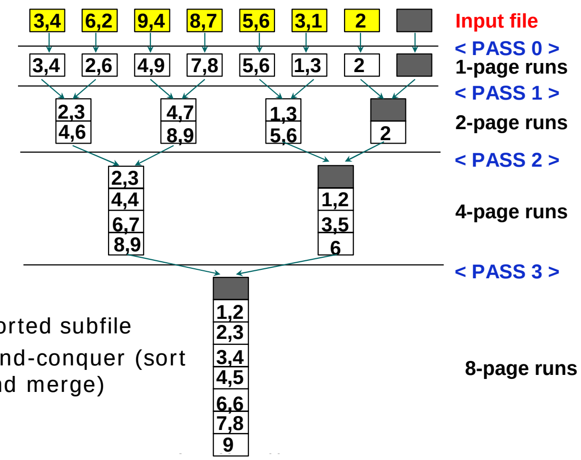
- Internal sorting
  - Sort a collection of records that fit within main memory can be done efficiently
  - There are a number of different sort algorithms that take $n\log_2 n$ time (i.e. $n \log_2 n$ comparisons)
    - Merge sort
    - Heap sort
- External sorting
  - Because of the large size of DB files, it may not be possible to fit all of the records to be sorted in main memory
  - The major cost consideration therefore is to reduce disk accesses while sorting (rather than concentrating on the number of comparisons)

Two-Way External Merge Sort

- Pass 0: read a page, sort it, write it
  - Any in-memory sorting algorithm can be used to sort the records on a page
  - Only one buffer page is used
- Pass 1, 2, … : read in and merge pairs of runs from the previous pass
  - Three buffer pages used
  - Output buffer is forced to disk one page at a time



Two-Way External Merge Sort (Cont.)



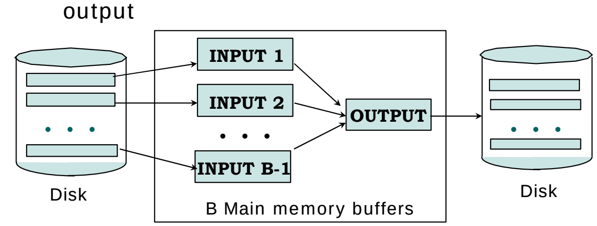
- run: a sorted subfile
- Divide-and-conquer (sort subfiles and merge)

Two-Way External Merge Sort (Cont.)

- I/O cost
  - N pages in the file: # of passes = $1 + \lceil \log_2 N \rceil$
  - In each pass, we read each page in the file, process it, and write it out: 2 disk I/Os per page, $2N$ disk I/Os per pass
  - So total cost = $2N\,(1 + \lceil \log_2 N \rceil)$ I/Os
- Only 3 buffer pages in memory are used, even if more buffer space is available
  - This simple algorithm does not utilize memory effectively!

General External Merge Sort

- To sort a file (N pages) using B buffer pages
  - Pass 0: read in B pages at a time and sort
    - Produce $\lceil N / B \rceil$ sorted runs of B pages each (the last run may contain fewer pages)
    - Use B buffer pages
  - Pass 1, 2, … : (B-1)-way merge
    - B-1 buffer pages for input; 1 buffer page for output

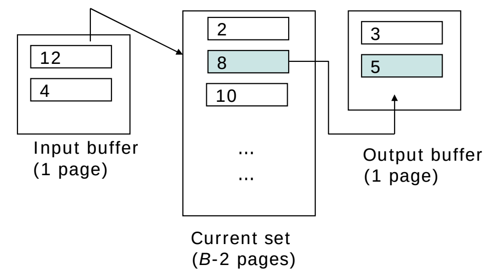


General External Merge Sort (Cont.)

- I/O cost
  - # of passes = $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
  - Cost = $2N * (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$ I/Os
- CPU cost of a multi-way merge can be greater than that for a two-way merge
- E.g., to sort a 108-page file with 5 buffer pages
  - 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: $\lceil 6/4 \rceil$ = 2 sorted runs, 80 pages & 28 pages
  - Pass 3: $\lceil 2/4 \rceil$ = 1 sorted file of 108 pages

Replacement Sort

- Can produce runs of about $2B$ sorted pages long, on average, in pass 0
  - Longer run means fewer passes



Replacement Sort (Cont.)

- To sort a file in ascending order of search key k
  - Read in pages of the file, until the input buffer and current set is full
  - Repeatedly pick the tuple in the current set with the smallest k that is $\geq$ the largest k in the output buffer, append it to the output buffer; move a new tuple from the input buffer to the current set
  - Read in the next page of file, if all tuples in the input buffer have been consumed
  - Write out the output buffer to extend the current run if it is full, or, as the last page if every tuple in current set is smaller than the largest tuple in the output buffer
  - Start a new run and continue the cycle if the current run is finished

I/O Costs Revisited

- The cost metric used so far (disk accesses) ignores two important details
  - Accessing several (consecutive) pages with a single I/O request may be much cheaper than accessing the same number of pages with independent requests: blocked I/O
  - I/O is only part of the costs, there are (non-trivial) CPU costs as well: double buffering

Blocked I/O

- The disk system supports reading / writing of a block of contiguous pages in a single I/O request
- We can make each buffer (input/output) be a block of *b* pages, *b* is blocking factor
  - One block per input run; one block for the output of the merge
  - Reduced fan-in during merge passes
    - $F = \lfloor B/b \rfloor - 1$ runs are merged in a pass
    - # of passes = $1 + \lceil \log_F \lceil N / B \rceil \rceil$
  - In practice, main memory sizes are large enough that most files are still sorted in 2-3 passes, even with blocked I/O

Blocked I/O (Cont.)

- I/O cost

  = (# of block I/Os) × (cost of a block I/O)
  - # of block I/Os

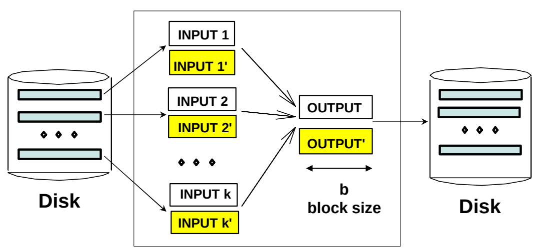
    = (# of page I/Os) ÷ (block size)

    = (# of passes) × (# of pages in the file) ÷ (block size)
  - Cost of a block I/O

    = seek time + rotational delay for 1st page + transfer time for the block

Double Buffering

- To reduce CPU wait time for I/O request to complete
  - Allocate an additional block (double block) to every input / output buffer
  - Tuples are pre-fetched into double blocks



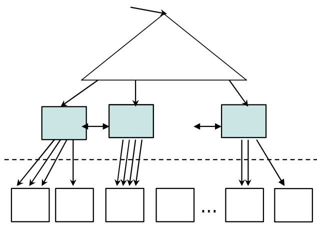
B buffer pages, k-way merge

B+ Trees for Sorting

- Scenario: table to be sorted has a B+ tree index on sorting column(s)
- Idea: retrieve records in search key order by traversing the leaf pages of B+ tree
- Is this a good idea?
- Cases to consider:
  - B+ tree is clustered: good idea!
  - B+ tree is unclustered: could be a very bad idea!

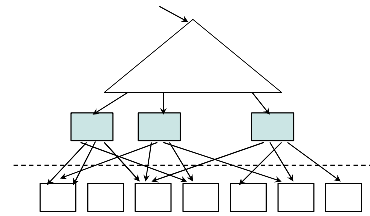
B+ Trees for Sorting (Cont.)

- Clustered B+ tree for sorting
  - Alternative 1: root to the left-most leaf, then retrieve all leaf pages
  - Alternative 2: additional cost of retrieving data records (each data page is fetched just once)
  - Always better than external sorting!



B+ Trees for Sorting (Cont.)

- Unclustered B+ tree for sorting
  - If Alternative (2) for data entries, each rid in the leaf page may point to a different data page
  - Worst-case cost = (cost of retrieving leaf pages to get data entries) + (# of data records)
  - Sorting by an unclustered index is inferior to external sorting



Summary

- External sorting is important; DBMS may dedicate part of buffer pool for sorting
- External merge sort minimizes disk I/O cost
  - Pass 0: produce sorted runs of size *B* (# of buffer pages); later passes: merge runs
  - # of runs merged at a time depends on *B* and block size
  - Larger block size means less I/O cost per page, but smaller # of runs merged
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad