CSE 541: Database Systems I

External Sorting

Slides Credit: Tianzheng Wang

Sorting

A very common and widely used operation in DBMS

- User may want the result in some order
  - E.g., the ORDER BY clause in SQL
- Sorting records is the first step in B+-tree bulk loading
- Sorting can be used to eliminate duplicate records
  - E.g., the DISTINCT, GROUP BY clauses in SQL
- Some very widely used join algorithms require a sorting step (i.e., input relations need to be sorted)

External Sorting

- Data is way larger than memory size
- Need to be able to sort a large amount of data using a small amount of memory, efficiently

Aside: the OS gives the applications the illusion that the application has practically unlimited “virtual” memory

- Backed by limited physical memory
  - E.g., Application sees 2^64 address space, on top of 8GB of memory
- So how about using OS Virtual Memory?
  - Not good – it does not know the workload and will generate many random I/Os, making the algorithm very slow

  ➔ Need more clever algorithms that can work with data that exceeds memory size efficiently, aka “out-of-core” algorithms

Out-of-Core Algorithms

Two main patterns:

- Single-pass streaming of data through main memory
- Divide-and-conquer
  - Divide data into smaller memory-sized chunks and process them in memory

Single-Pass Streaming

- Suppose we have a very big table stored in disk
- Goal: apply a function f(x) to each record x and store the result back to the table in disk
- Desired: use the minimum amount of memory and do disk read/writes as little as possible
- Solution
  - Read table records in chunks from disk to an input buffer in memory
  - Apply the function and place the new record in an output buffer
    - Note: record size may change after applying f(), e.g., if f() does compression
  - Whenever the output buffer is full, write it out to disk
    - Each flush is a full page of I/O
  - After the input buffer is consumed, load another chunk of records

  ➔Need only two buffers in total, data is streamed through memory only once
- New problem: when output buffer is being flushed, the CPU sits idle

Overlapping I/O and Computation

Separate computation and I/O into two threads so that I/O and computation can overlap

- I/O can be local storage I/O or network operations
- A useful design in many systems, not limited to DBMS

Main thread:

- Read input buffer to apply function f(x), write result to output buffer

I/O thread:

- Flush ("drain") the output buffer; load new chunk to input buffer
- While I/O thread is blocked on I/O, the main thread can continue to process more records in the input buffer

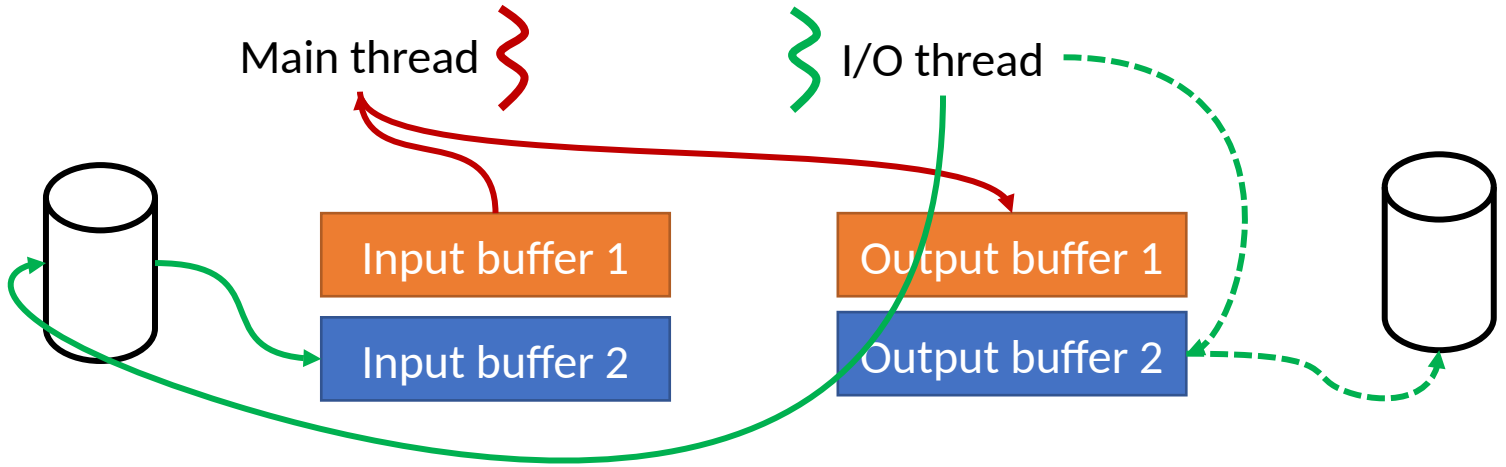
Problem: a buffer cannot be written and flushed at the same time

- I.e., when the output buffer is being flushed by the I/O thread, the main thread cannot write to it
- Solution: double buffering

Double Buffering

Use two input buffers, two output buffers. Switch them when I/O is needed on them

- While main thread is working on input buffer 1 and output buffer 1, I/O thread works on the other pair of buffers



- Main thread switches to work on input buffer 2 and output buffer 2, while I/O thread starts to load new data into input buffer 1 and flush output buffer 1

Aside: this optimization applies to logging, too

- Write one log buffer B1, switch to B2 and let flusher drain B1

Simple Two-Way Merge Sort

Merge sort: break file into smaller sub-files (runs), sort sub-files and merge the sorted runs to obtain final result

- Pass 0: Read each page, sort it, write it out
  - Once in memory, sort the data on a page using some in-memory sorting algorithm (e.g., quick sort)
  - Write out the sorted buffer (a “sorted run”)
  - Need one buffer in memory
- Pass 1
  - Read two runs produced by Pass 0 each time, merge them to produce a new sorted run (2x size of runs in Pass 0)
    - Repeat: pick the smaller value from the runs and put it in the output buffer
  - Need three buffers (two for input, one for output)

Simple Two-Way Merge Sort

- Pass 2
  - Read two runs produced by Pass 1 each time, merge them to produce a new sorted run (2x size of runs in Pass 1)
  - Need three buffers
- Repeat with more passes until one sorted run is produced

For an input file with 2^k pages:

Pass 0: produces 2^k sorted runs, 1 page each

Pass 1: produces 2^{k-1} sorted runs, 2 pages each

Pass 2: produces 2^{k-2} sorted runs, 4 pages each

…

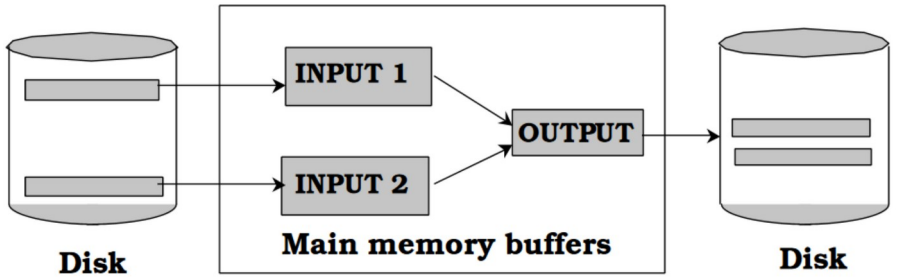
Pass k: produces one sorted run, with all the 2^k pages

Simple Two-Way Merge Sort

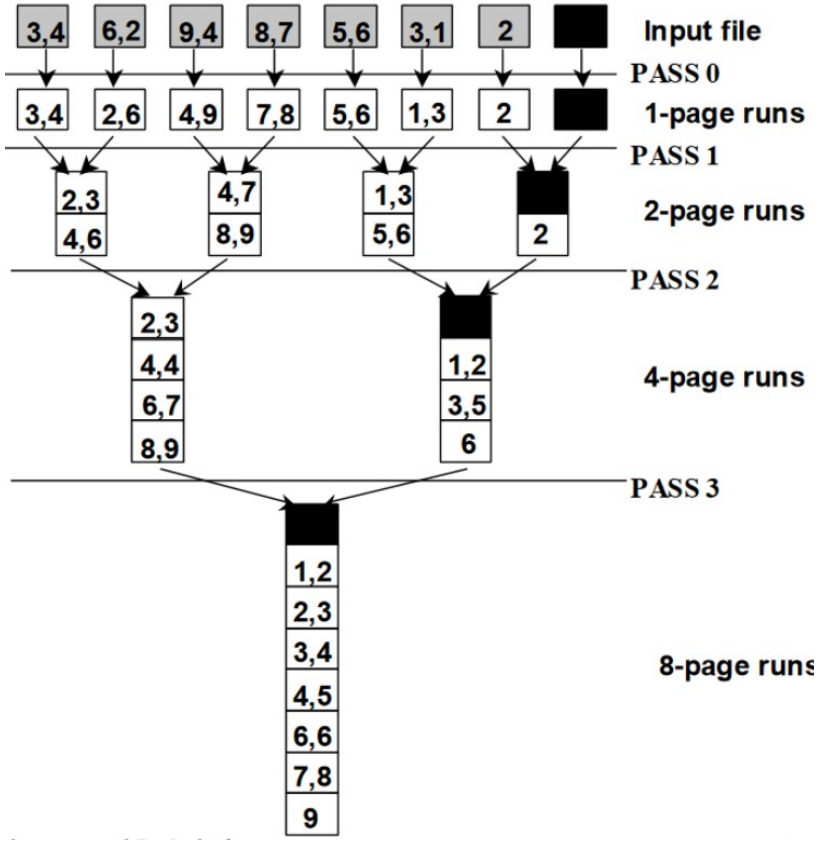
- Each pass reads and writes each page in the file
- N pages in the file

  ➔ number of passes = $\lceil \log_2 N + 1 \rceil$
- Total cost:

  $2 * N * (\lceil \log_2 N + 1 \rceil)$
- Need three buffers in total



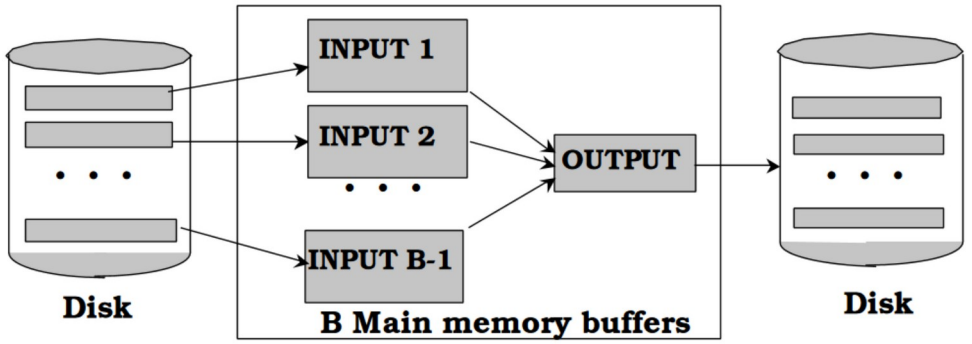
Example with 7 pages:



General External Merge Sort

Goal: utilize more available memory buffers

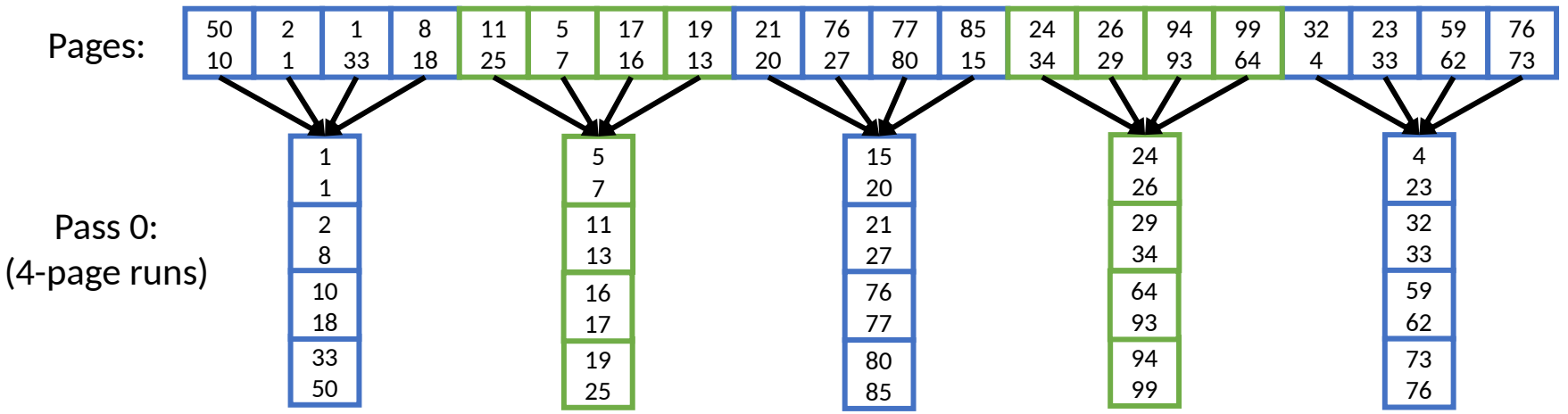
- Sort a file with N pages using B buffer pages
- Pass 0: write out more output pages each time
  - Using B buffer pages, produce $\lceil N/B \rceil$ sorted runs, each run has B pages
- Pass 1, 2, … merge many runs at the same time
  - Using B – 1 buffer pages, do a (B-1)-way merge (merge B-1 runs)
  - Each buffer is for loading a page of a run generated in the previous pass
    - Keep picking the smallest record across all B-1 buffers
    - Put it on the output buffer (flush if full)



General External Merge Sort

Example: sort a file of 20 pages, 4 buffers

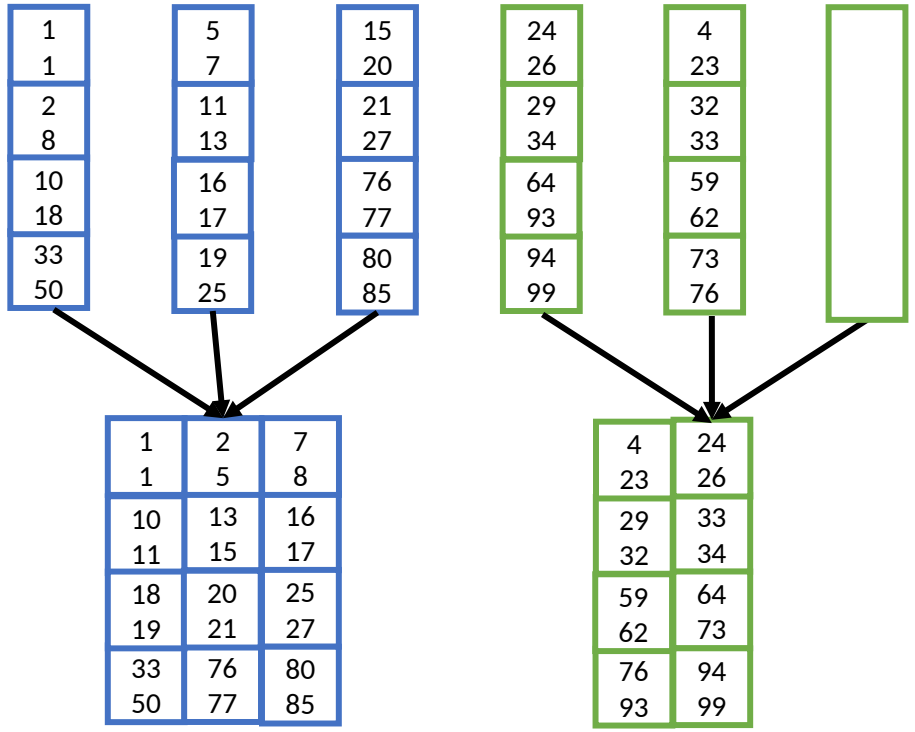
- Pass 0: write out more output pages each time
  - Using B buffer pages, produce N/B sorted runs, each run has B pages
  - B = 4 (4 buffers)
  - Read in Pages 0, 1, 2, 3, produce a sorted run
  - Read in Pages 4, 5, 6, 7, produce a sorted run
  - … continue to generate all runs



General External Merge Sort

- Pass 1, 2, … merge many runs at the same time
  - Using B – 1 buffer pages, do a (B-1)-way merge (merge B-1 runs)
  - B = 4 ➔ Use 3 buffers, do a 3-way merge
    - One buffer per run, choose the smallest element to put on output buffer

Pass 0:
(4-page runs)



Pass 1:
(12-page runs)

Total number of passes:

$3 = 1 + \lceil \log_{4-1}(20/4) \rceil$

$= 1 + \lceil \log_3 5 \rceil$

$= 1 + 2$

$$1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$$

➔ Continue with pass 2 and finish

Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2 * N * (number of passes)
- Example: sort a 108-page file with 5 buffer pages
  - ➔ $1 + \lceil \log_4 22 \rceil = 1 + 3 = 4$ passes
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs, 5 pages each (3 pages in the last run)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs, 20 pages each (8 pages in the last run)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: file sorted with 108 pages

Number of Passes in External Merge Sort

# of pages	B = 3	B = 5	B = 9	B = 17	B = 129	B = 257
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
10000	13	7	5	4	2	2
100000	17	9	6	5	3	3
1000000	20	10	7	5	3	3
10000000	23	12	8	6	4	3
100000000	26	14	9	7	4	4
1000000000	30	15	10	8	5	4

Memory Requirement

Q: Is it possible to sort a file/table in two passes? Or what is the maximum table size can we sort using two passes with B buffers?

- Each sorted run after Pass 0 has B pages
  - Note: page size = buffer size
- Pass 1 can merge B-1 sorted runs generated in Pass 0

A: Maximum size = B * (B – 1) pages

- B * (B – 1) close to B * B ➔ roughly we can sort a file of N pages using sqrt(N) buffers

Using B+-Tree for Sorting

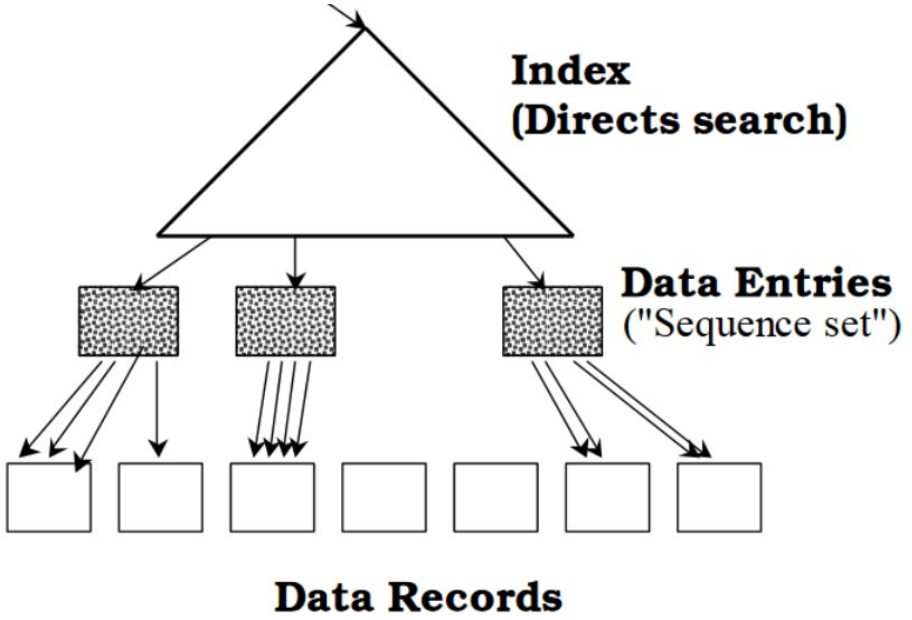
Scenario: Table to be sorted has B+ tree index on sorting column(s)

Idea: Retrieve records in order from index by traversing leaf pages

- Depends on whether the B-tree is clustered

Using a clustered B-tree:

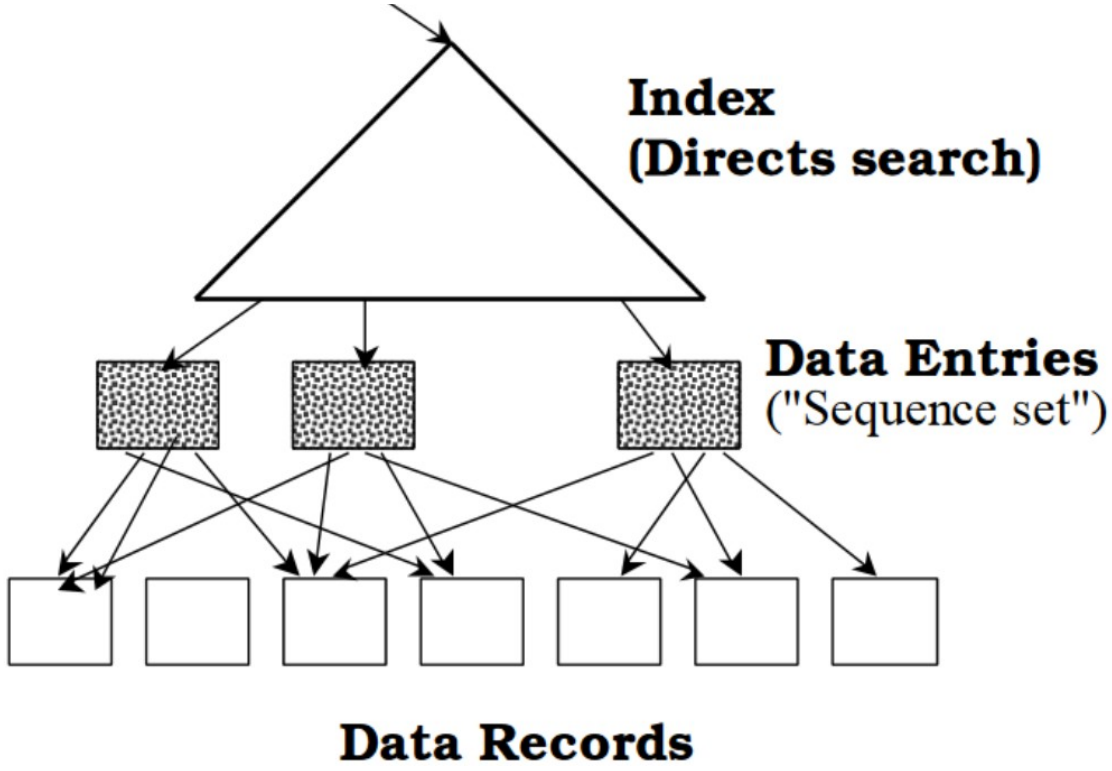
- Alternative 1 (data in leaves)
  - Cost = traverse to left-most leaf + retrieve all leaf nodes
- Alternative 2/3 (RIDs in leaves)
  - Add extra cost to access records



Using B+-Tree for Sorting

Using an unclustered B-tree:

- Alternative 2 (RIDs in leaves)
  - May need one I/O per record



Summary

- Uses of sorting in DBMS
  - Join algorithms, removing duplicates, required by user, etc.
  - Often “out-of-core” – need to sort data files larger than memory
- Out-of-core algorithms
  - Streaming – read data into memory once, and (possibly modify it) and write it out
  - Double buffering – overlap I/O and computation
    - Common strategy for building high-performance data-intensive systems
- Simple two-way and general merge sort algorithms
  - Break a big file into smaller chunks and merge them
  - Pass 0 generates sorted runs, further passes read in the sorted runs using given memory buffers and generate sorted output
- B-trees can be used for sorting by scanning leaf nodes