Fall 2013

EXTERNAL SORTING
(CH. 13 IN THE COW BOOK)

Motivation for External Sort

- Often have a large (size greater than the available main memory) that we need to sort.
- Why are we sorting:
  - Query processing: e.g. there are sort-based join and aggregate algorithms
  - Bulkload B+-tree: recall you had to sort the data entries in the leaf level for this.
  - One can specify ORDER BY in SQL, which sorts the output of the query
  - …

Problem Statement

- Given **M** memory pages, and a relation of size **N** pages, where N > M, sort R on a **sort key**, to produce an output relation R' that is sorted on the sort key.
- Example: Sort the following table on zipcode

```
CREATE TABLE Tweets (
   uniqueMsgID INTEGER,        -- unique message id
   tstamp      TIMESTAMP,      -- when was the tweet posted
   uid         INTEGER,        -- unique id of the user
   msg         VARCHAR (140),  -- the actual message
   zip         INTEGER,        -- zipcode when posted
   retweet     BOOLEAN         -- retweeted?
   );
```

- Another example: SELECT * FROM Tweets ORDER BY zip, retweet

Note the sort key can be composite

Goal of a good sort algorithm

- Sort efficiently!
- Sort well!
  - Able to sort large relations with "small" amounts of main memory
- What does sort efficiently mean:
  - Minimize the number of disk I/Os
  - Try using sequential I/Os rather than random I/Os
  - Minimize the CPU costs
  - Overlap I/O operations with CPU operations

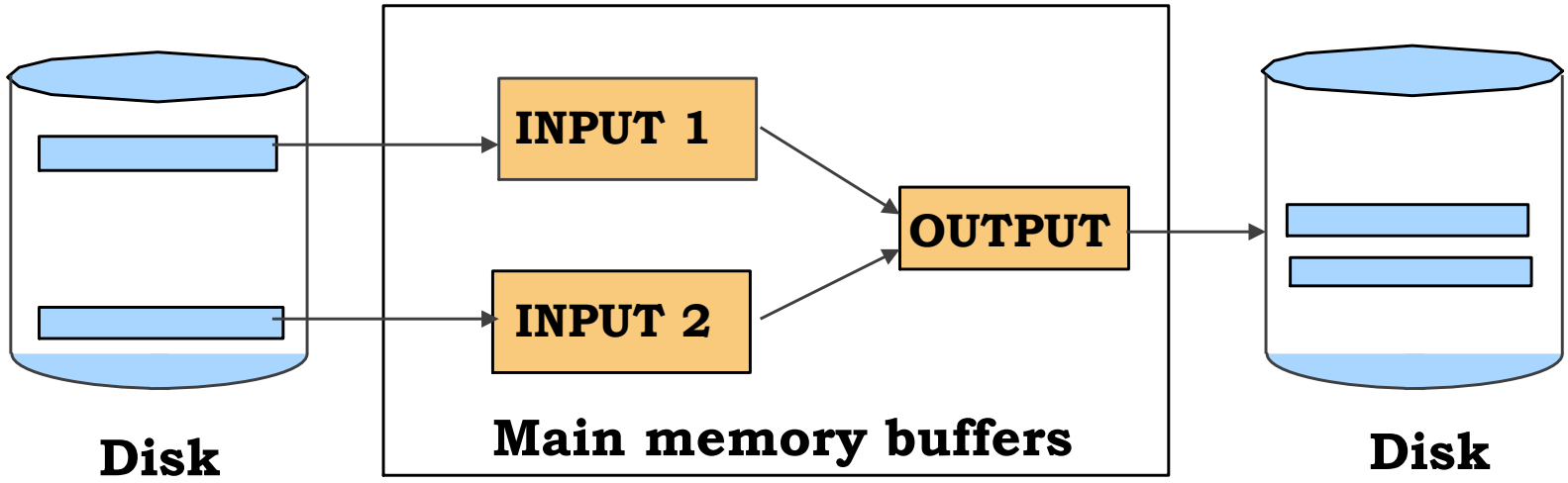
Where does the memory come from?

Quick note: Sorting is very important in MapReduce. The reducer expects data to arrive in sorted order from the mappers.

2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it (a **run**).
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.

Algorithms for sorting in memory?



Two-Way External Merge Sort

- Read & write entire file in each pass
- N pages, # passes =

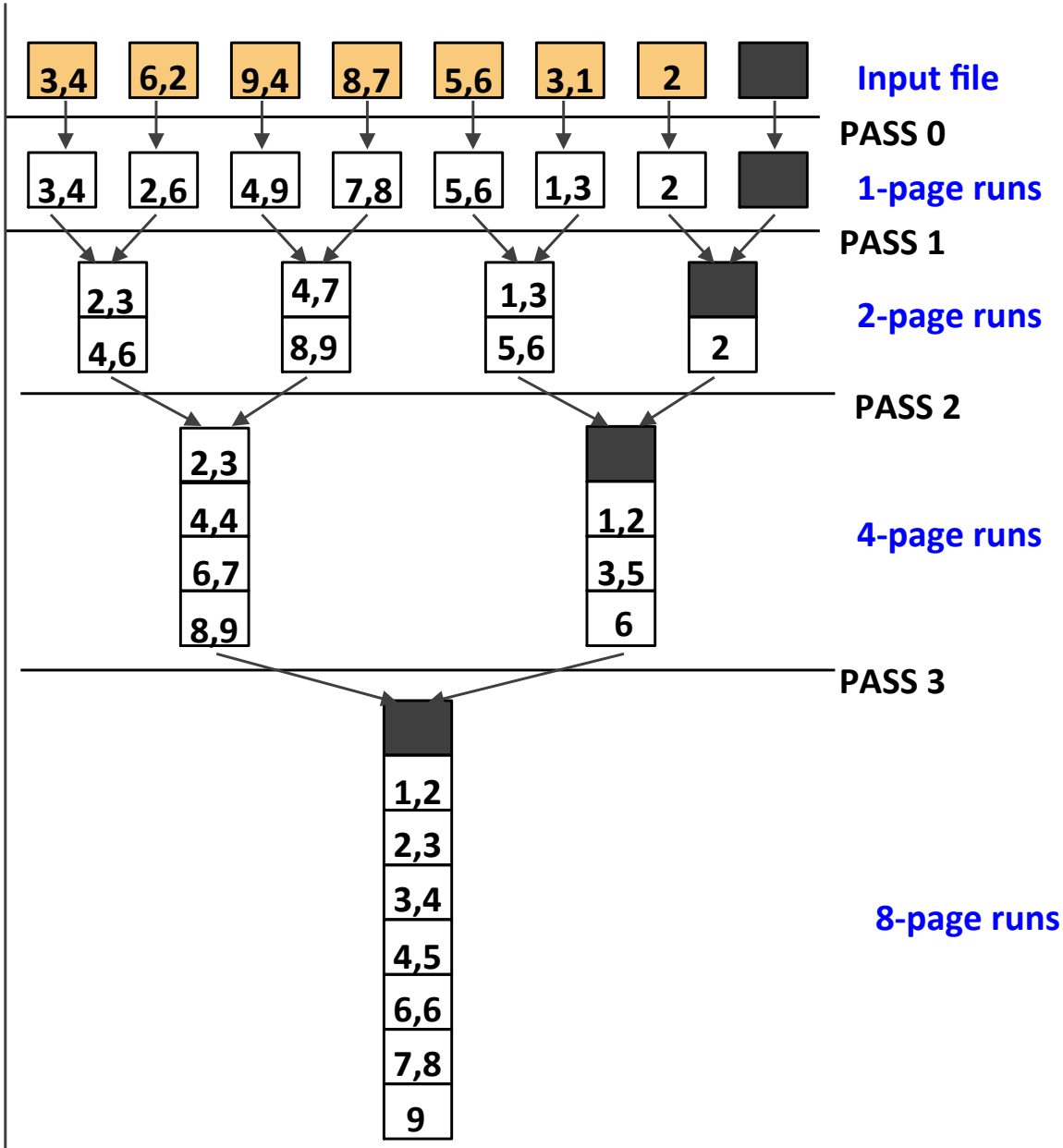
$$\lceil \log_2 N \rceil + 1$$

- So total cost is:

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

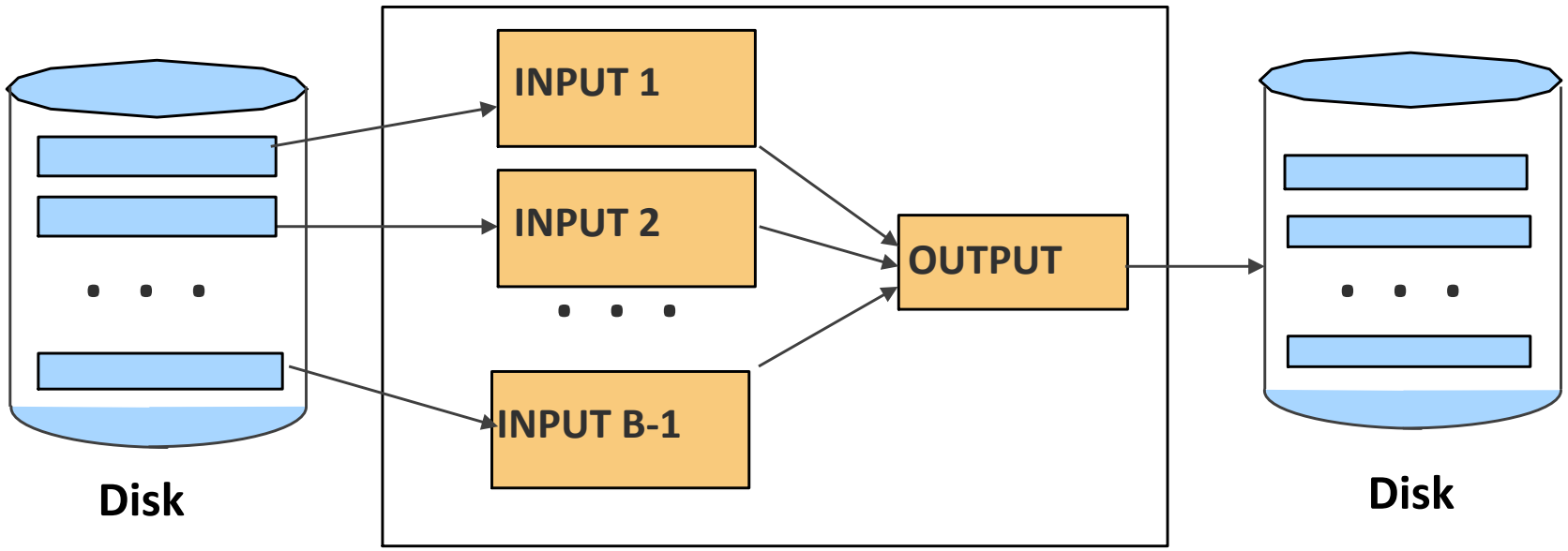
- **Divide and conquer**

How can we utilize more than three buffer pages?



General External Merge Sort

- Sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages (run size = B pgs). Produce $\lceil N/B \rceil$ sorted runs of *B* pages each.
  - Pass 2, 3, …: merge ***B-1*** runs.



B-1 way merge.
Total buffer pages: B

Where are the main memory buffer pages allocated?

Cost of External Sort Merge

- # passes =
- I/O Cost = # passes * 2 N
- Consider sorting a file with a 1000 pages, using 11 buffer pages.
  - At the end of the first pass, we have runs of size pages
  - Next pass produces runs of size pages each
  - The next pass

Number of Passes of External Sort

N (# of pages)	B=3	B=17	B=257
100	7	2	1
10,000	13	4	2
1,000,000	20	5	3
10,000,000	23	6	3
100,000,000	26	7	4
1,000,000,000	30	8	4

32K pg size, 32TB relation

@1ms per read, 1111 hours = **46 days!**

Internal Sort Algorithm: Replacement Sort

Size of the buffer pool?

Example: M = 2 pages, 2 tuples per page.
Input Sequence: 10, 20, 30, 40, 25, 35, 9, 8, 7, 6, 5, …

1.		In-memory 10, 20, 30, 40
2.	Read 25, Output 10.	In-memory: 20, 25, 30, 40
3.	Read 35, Output 20.	In-memory : 25, 30, 35, 40
4.	Read 9, Output 25.	In-memory : 9, 30, 35, 40
5.	Read 8, Output 30.	In-memory : 8, 9, 35, 40
6.	Read 7, Output 35.	In-memory : 7, 8, 9, 40
7.	Read 6, Output 40.	In-memory : 6, 7, 8, 9
8.	Read 5, Flush output, Start new run.	In-memory …
	On Disk: 10, 20, 25, 30, 35, 40	

Average length of a run in replacement sort is *2M*

Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.
- An alternative is **replacement sort**, which is also called tournament sort or heapsort
  - `Top`: Read in ***M*** pages of the relation R
  - `Output`: move smallest record to output buffer
  - Read in a new record *r*
  - insert *r* into "sorted heap"
  - if *r* not smallest, then `GOTO Output`
  - else remove *r* from "heap"
  - output "heap" in order; `GOTO Top`
- Worst-Case: What is min length of a run? How does this arise?
- Best-Case: What is max length of a run? How does this arise?
- Quicksort is faster, but longer runs often means fewer passes!

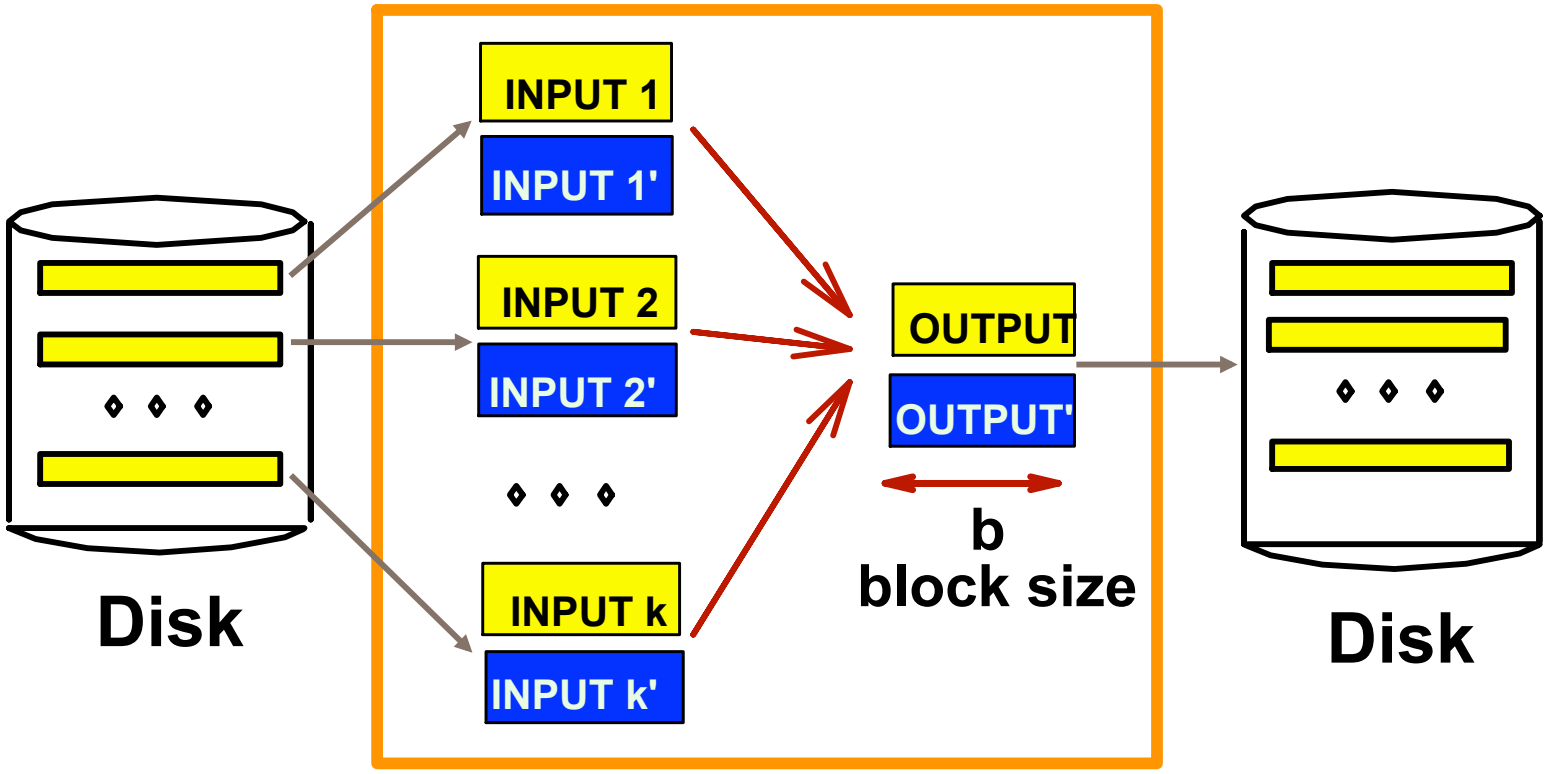
Blocked I/Os

- So far we reading/writing one page at a time, but we know that reading a **block** of pages sequentially is faster.
- Make each buffer (input/output) be a block of pgs.
  - Will reduce fan-out during merge passes! Side-effect?
  - Reduces per page I/O cost.
  - First Pass: Each run **2B** pages, $\lceil \mathbf{N/2B} \rceil$ runs (where **B** is the size of the buffer pool in #pages)
    - Which internal sort algorithm are we using?
  - Merge Tree Fanout: $\mathbf{F = \lfloor B/b \rfloor - 1}$, b is block size
  - # passes: $\lceil \mathbf{\log_F \ldots} \rceil \mathbf{+ 1}$
  - In practice, buffer pools are large, so most files are sorted in 2-3 passes

Double Buffering

Reduces response time.
What about throughput?

- Overlap CPU and IO processing
- *Prefetch* into shadow block.
  - Potentially, more passes; in practice, 2-3 passes.



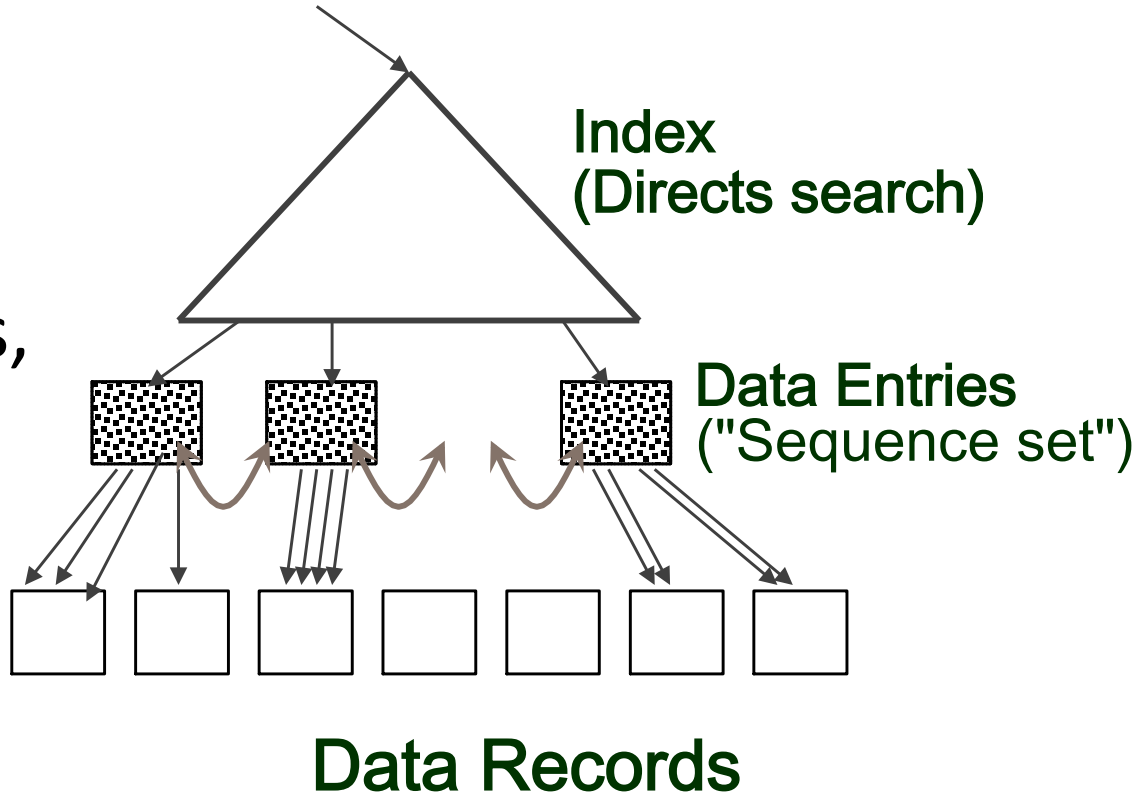
B main memory buffers, k-way merge

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!***

Clustered B+ Tree Used for Sorting

- Go to the left-most leaf, then retrieve all leaf pages
- If data entry has records, then we are done!
- If the data entries have **rid**s, each data page is fetched just once (since this is a clustered index)



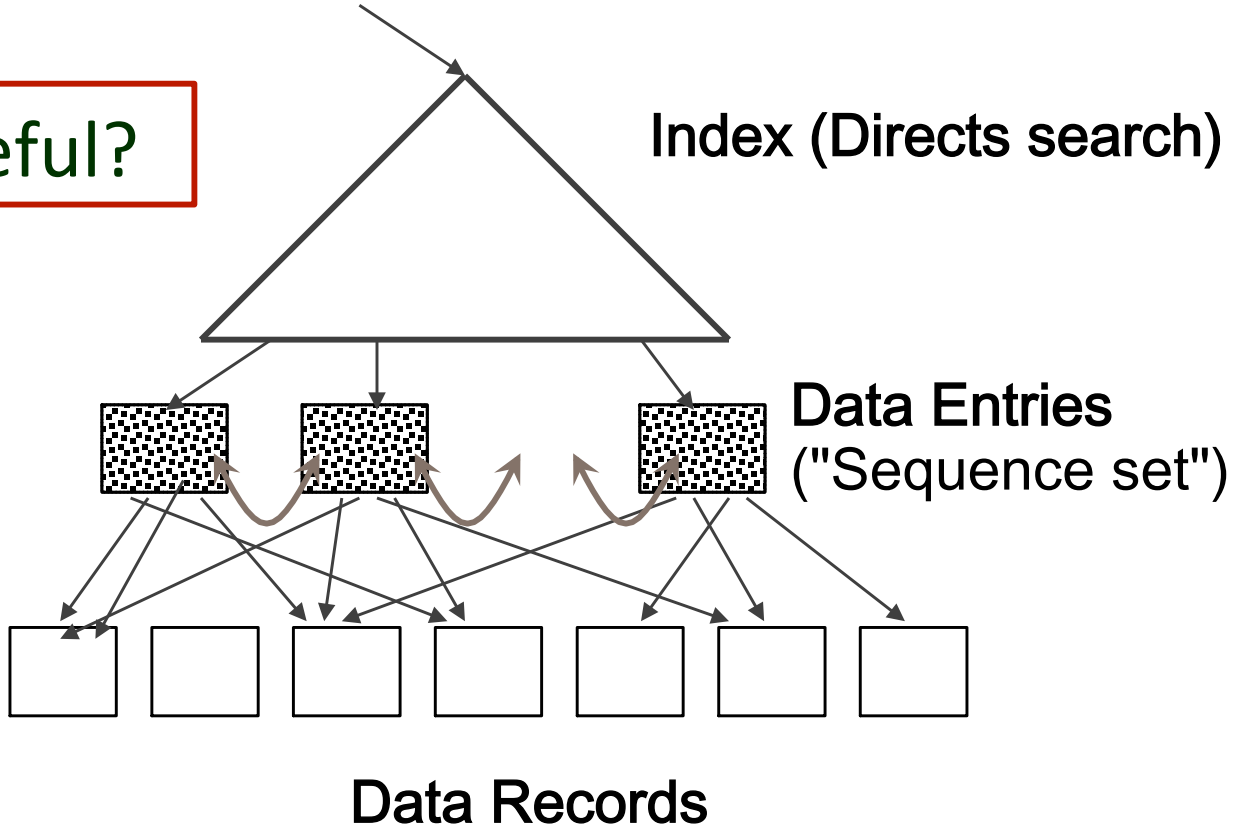
Faster than external sorting!

Why not scan the data file directly?

Unclustered B+ Tree Used for Sorting

- Unclustered B+-trees only have **rid**s in the data entries
- So, in general, one I/O per data record!

When can this be useful?



Sorting Records!

- Sorting is a competitive sport!
- See http://sortbenchmark.org/
  - Task is to sort 100 byte records.
  - Different flavors of metrics that people compete on.

  a) Sort at trillion records as fast as you can, using general purpose sorting code (Daytona) or code specialized just for the benchmark (Indy)

  b) Sort as many records as you can in a minute