Ch 13. External Sorting

Sang-Won Lee

http://icc.skku.ac.kr/~swlee

SKKU VLDB Lab.

(http://vldb.skku.ac.kr/)

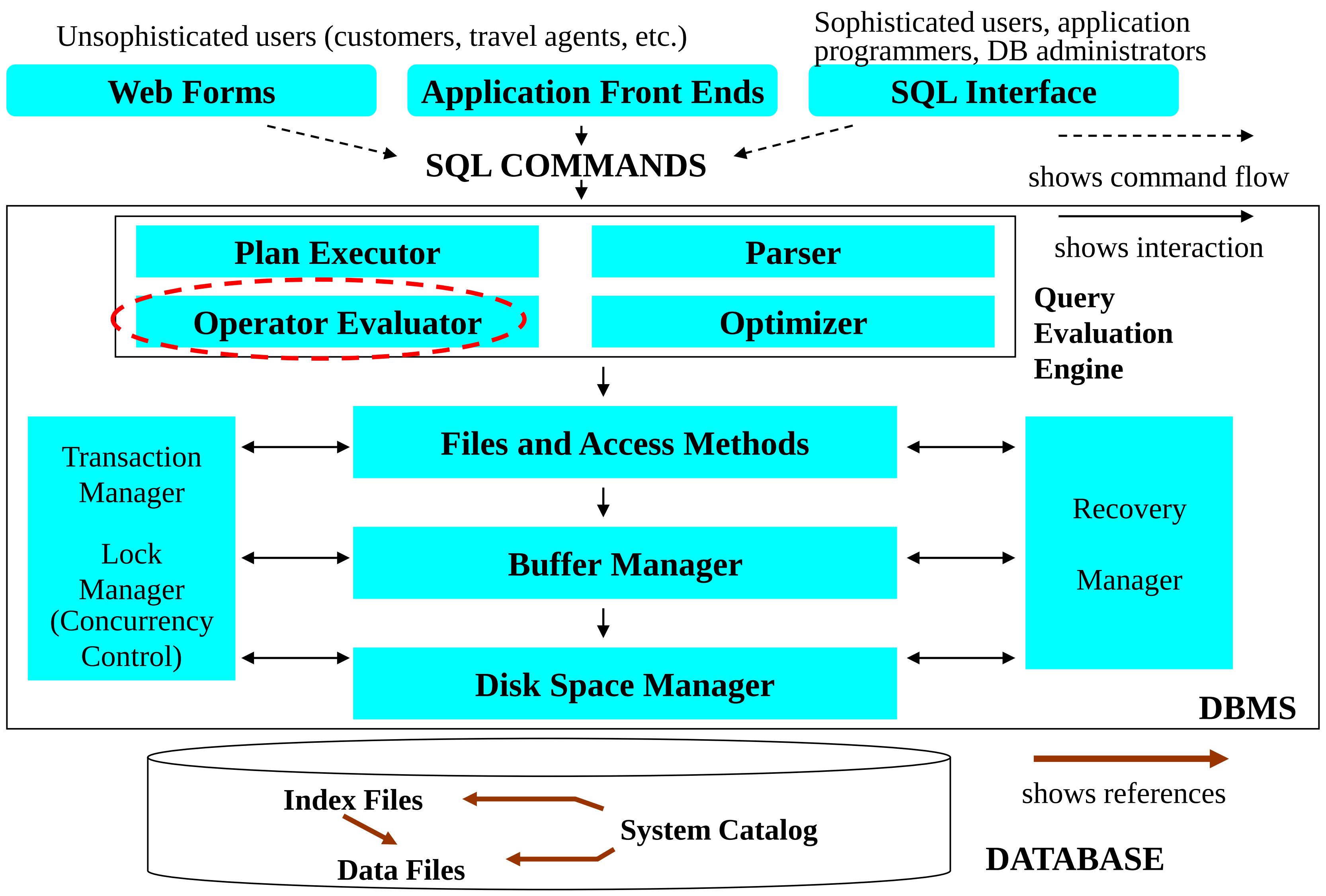


Figure 1.3 Anatomy of an RDBMS

What you should know from this chapter

- Why is sorting so important in a DBMS?
- In-memory sorting vs. external sorting
- External merge sort
  - Basic algorithm: “Load-Sort-Store”
  - Some optimizations: blocked & overlapped I/O
- B+ tree and sorting

13.1 Why Sorting?

From Knuth's book Sorting and Searching:

> Computer manufacturers of the 1960's estimated that more than 25 percent of the running time of their computers was spent on sorting, when all their customers were taken into account. In fact, there were many installations in which the task of sorting was responsible for more than half of the computing time. From these statistics we may conclude that either
>
> 1. there are many important applications of sorting, or
> 2. many people sort when they shouldn't, or
> 3. inefficient sorting algorithms have been in common use.

Computing has changed since the 1960's, but not so much that sorting has gone from being extraordinarily important to unimportant.

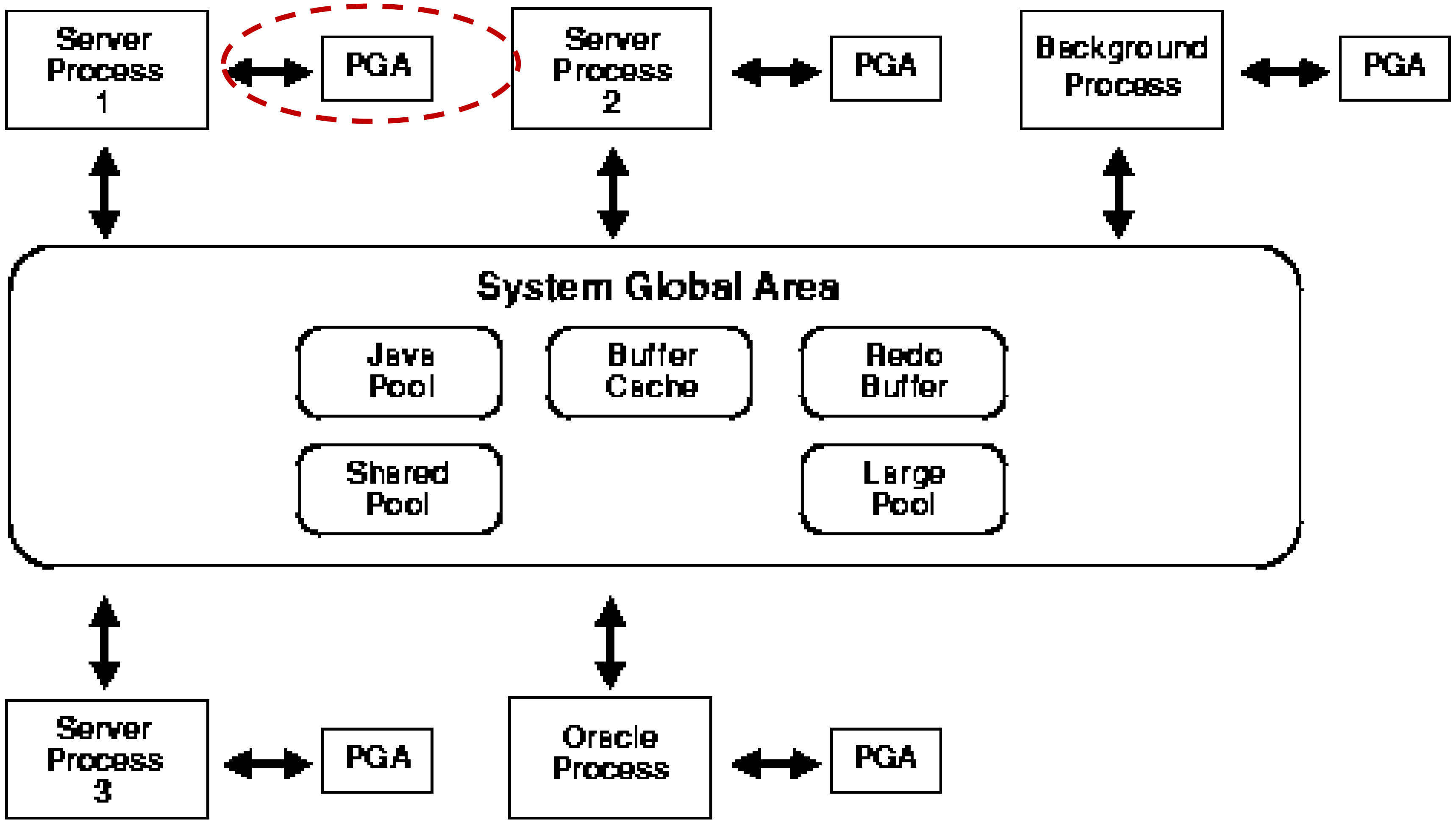
13.1 Why Sorting?

- A classic problem in computer science! Particularly in DB!!
  - Data requested in sorted order: order by
    - ✓ e.g., SELECT a, b, c FROM r ORDER BY a
  - Sorting is the first step in *bulk loading* B+ tree index.
  - *Sort-merge* join algorithm involves sorting.
  - And, alternative implementation for “distinct” and “group by”
    - ✓ vs. hashing

Sorting Problem

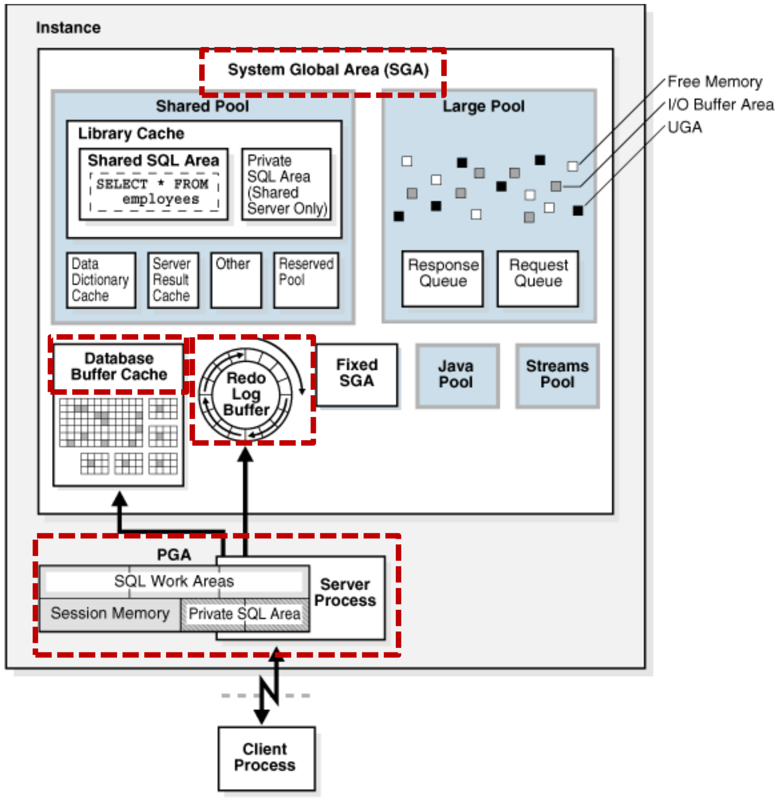
- Problem: e.g. sort 1Gb of data with 1MB of RAM.
  - NOTE: many commercial DBMSs, such as Oracle and DB2, use separate main memory, not buffer cache, for sorting!
    - ✓ e.g. Oracle’s sort_area_size in PGA (see next two slides for details)
- Let N = 2^s be the size of the input file to be sorted: N = # of pages
  - Assume this size is too large to fit in available memory
- Approaches
  - 2-way merge sort
  - N-way merge sort
  - Three optimizations to reduce # of IOs

Oracle Memory Structure

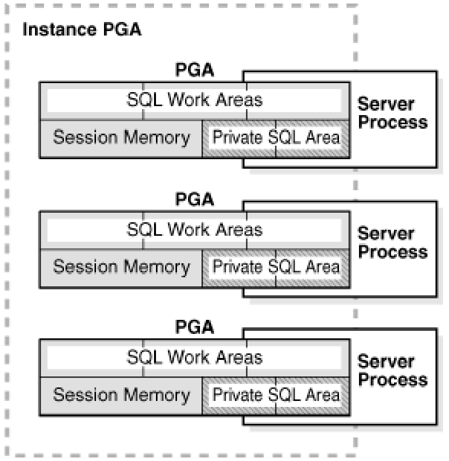


Oracle 19C: Memory Architecture

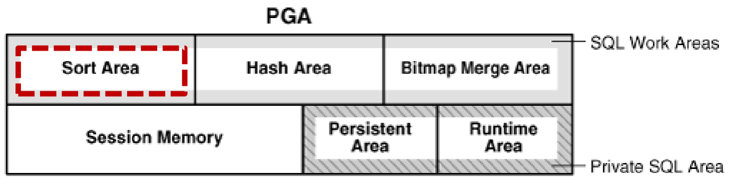
- https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/memory-architecture.html#GUID-913335DF-050A-479A-A653-68A064DCCA41



Oracle DB Memory Structure

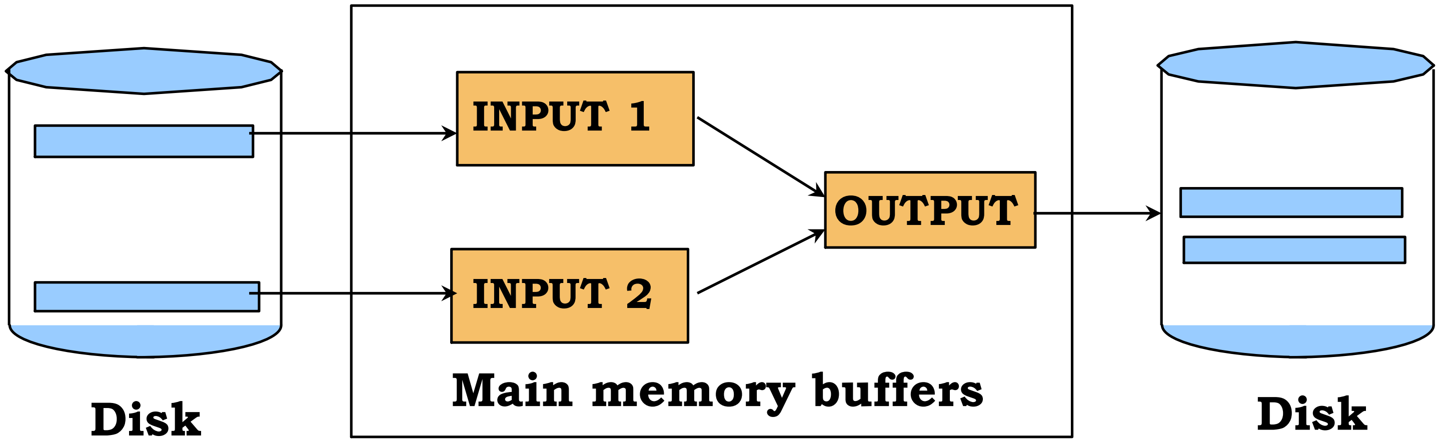


Oracle Instance PGAs
(Program Global Area)



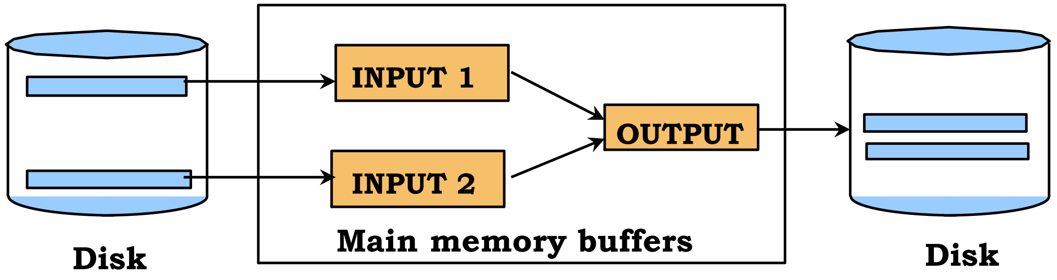
PGA Content

13.2 2-Way External Merge Sort (with 3 Buffers)



2-Way External Merge Sort

- Pass 0: using only 1 buffer page
  1. Read each of N pages page-by-page
  2. Sort the records on each page individually (using an in-memory sort algorithm)
  3. Write the sorted page to disk
     - ✓ Each sorted page is called as a **run**
  - # of runs = 2^s



- Pass 1, 2, 3, …, etc.: using three pages
  1. Read 2 runs generated in Pass 0 into 2 input buffers
  2. Merge their records into the other output buffer
  3. Write the new merged buffer to disk (page-by-page)
  - # of runs = 2^(s-i) , where *i* is the pass number

2-Way External Merge Sort: Example

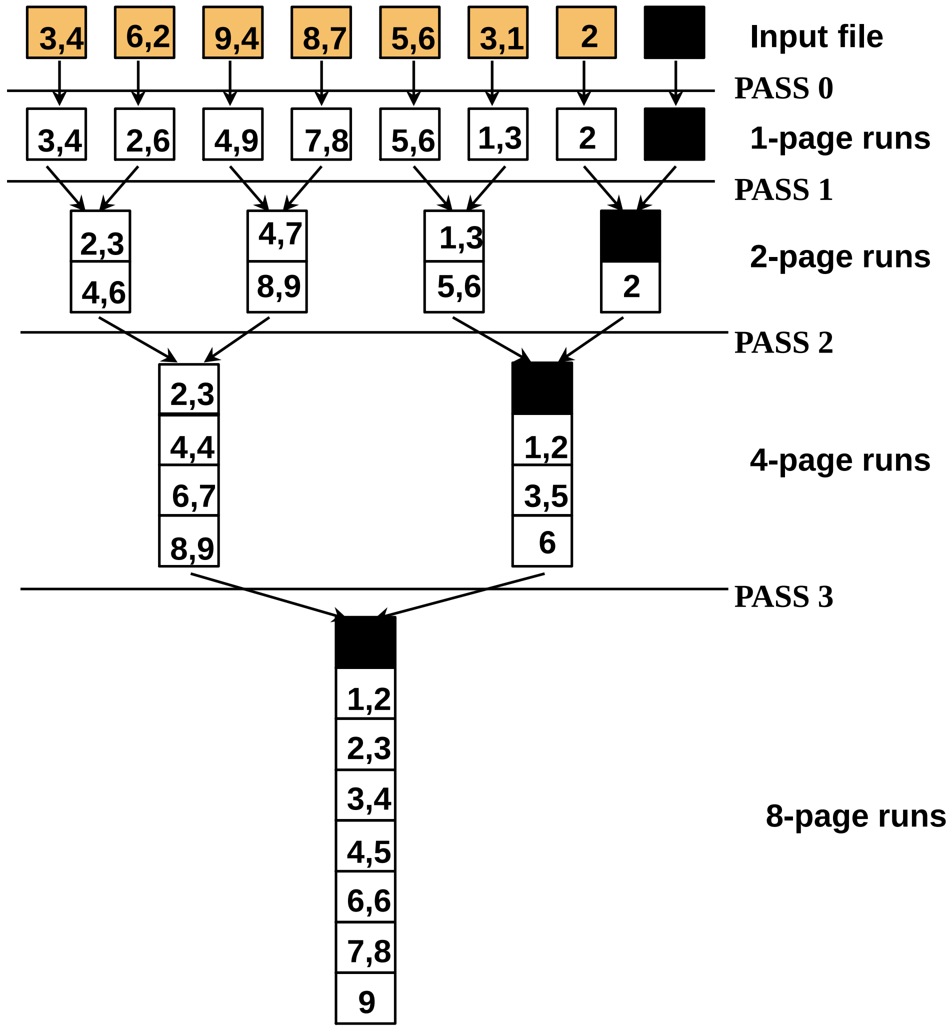
- Each pass, we read + write each page in file.
- N pages in the file ➔ the number of passes

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

- So total cost is:

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

- Idea: "divide and conquer"
  - "sort subfiles and merge"



2-Way External Merge Sort: Algorithm

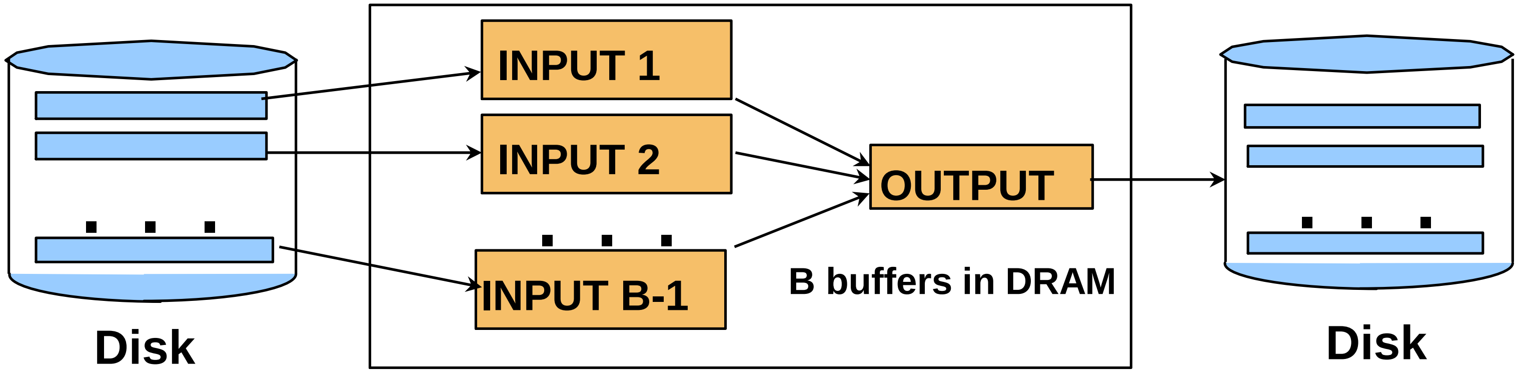
```
Algorithm: two-way -merge-sort(f ,N, )
Input: Number of pages N = 2^s in input file f
Output: sorted file “run_s_0” written to disk (side-effect)

// Pass 0: write 2s sorted single-page runs
for each page r in 0 . . . 2s − 1 do
        pp = pinPagef (r );
        f0 = createFile(“run_0_r”);
        sort the records on page pointed to by pp;
        write page pointed to by pp into file f0;
        closeFile(f0);
        unpinPagef (r, false);
end for;
```

```
// Passes 1 . . . s
for n in 1 . . . s do
// pair-wise merge of all runs written in pass n-1
  for r in 0 . . . 2s−n − 1 do
        f1 = openFile(“run_(n − 1)_(2r )”);
        f2 = openFile(“run_(n − 1)_(2r + 1)”);
        f0 = createFile(“run_n_r”);
        merge the records in files f1, f2,
        and write merged result to f0
        (using 3 pages of buffer space)
        closeFile(f0);
        deleteFile(f1);
        deleteFile(f2);
  end for;
Enf for;
I N.B. The run file “run_n_r” contains the r-th run of Pass n.
```

13.2 General External Merge Sort (with *B* pages)

- **More than 3 buffer pages. How can we utilize them?**
- To sort a file with ***N*** pages using ***B*** buffer pages:
  - Pass 0: use *B* buffer pages.
    - ✓ Reduce # of initial runs
    - ✓ Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
  - Pass 1, 2, …, etc.: merge *B-1* runs
    - ✓ Reduce # of passes

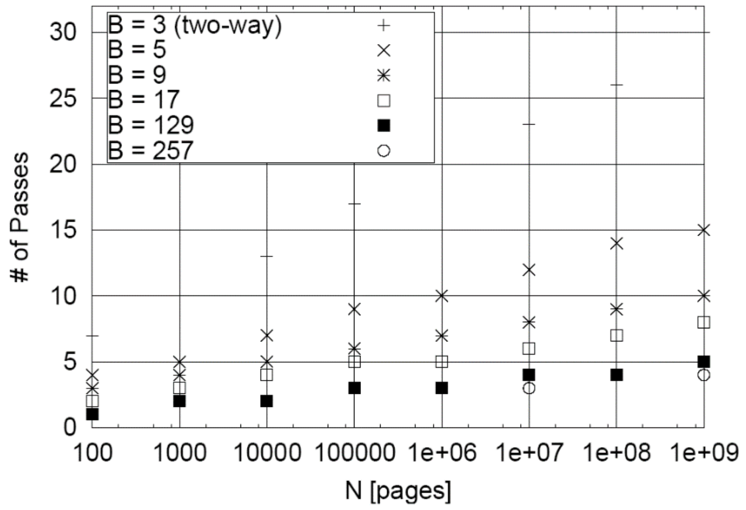


Cost of External Merge Sort

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

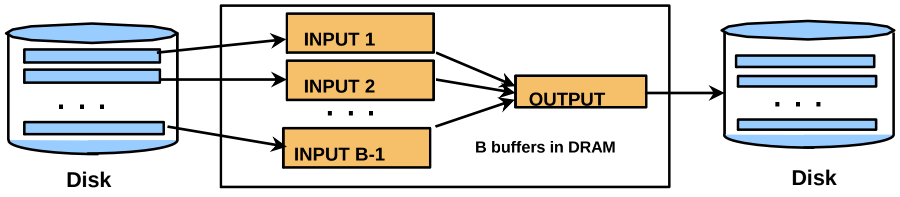
Number of Passes of External Sort

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



External Merge Sort: Comparison Cost

- External sort reduces the I/O cost, but is **CPU-intensive**
- E.g. to pick the next record for output buffer, we have to do B-2 comparisons



Example (let $B - 1 = 4, \theta = <$):

$$\left\{\begin{array}{l} 087\ 503\ 504\ \dots \\ 170\ 908\ 994\ \dots \\ 154\ 426\ 653\ \dots \\ 612\ 613\ 700\ \dots \end{array}\right. \dashrightarrow 087 \left\{\begin{array}{l} 503\ 504\ \dots \\ 170\ 908\ 994\ \dots \\ 154\ 426\ 653\ \dots \\ 612\ 613\ 700\ \dots \end{array}\right. \dashrightarrow 087\ 154 \left\{\begin{array}{l} 503\ 504\ \dots \\ 170\ 908\ 994\ \dots \\ 426\ 653\ \dots \\ 612\ 613\ 700\ \dots \end{array}\right. \dashrightarrow$$

$$087\ 154\ 170 \left\{\begin{array}{l} 503\ 504\ \dots \\ 908\ 994\ \dots \\ 426\ 653\ \dots \\ 612\ 613\ 700\ \dots \end{array}\right. \dashrightarrow 087\ 154\ 170\ 426 \left\{\begin{array}{l} 503\ 504\ \dots \\ 908\ 994\ \dots \\ 653\ \dots \\ 612\ 613\ 700\ \dots \end{array}\right. \quad \dots$$

External Merge Sort: Comparison Cost

- An improvement for CPU-intensive comparison
  - To use a bottom-up **selection tree** to support merging

$$
087 \left\{ \begin{array}{l} 087 \left\{ \begin{array}{l} 087\ 503\ 504\ \ldots \\ 170\ 908\ 994\ \ldots \end{array} \right. \\ 154 \left\{ \begin{array}{l} 154\ 426\ 653\ \ldots \\ 612\ 613\ 700\ \ldots \end{array} \right. \end{array} \right. \dashrightarrow 087\ 154 \left\{ \begin{array}{l} 170 \left\{ \begin{array}{l} 503\ 504\ \ldots \\ 170\ 908\ 994\ \ldots \end{array} \right. \\ 154 \left\{ \begin{array}{l} 154\ 426\ 653\ \ldots \\ 612\ 613\ 700\ \ldots \end{array} \right. \end{array} \right. \dashrightarrow 087\ 154\ 170 \left\{ \begin{array}{l} 170 \left\{ \begin{array}{l} 503\ 504\ \ldots \\ 170\ 908\ 994\ \ldots \end{array} \right. \\ 426 \left\{ \begin{array}{l} 426\ 653\ \ldots \\ 612\ 613\ 700\ \ldots \end{array} \right. \end{array} \right.
$$

$$
\dashrightarrow 087\ 154\ 170\ 426 \left\{ \begin{array}{l} 503 \left\{ \begin{array}{l} 503\ 504\ \ldots \\ 908\ 994\ \ldots \end{array} \right. \\ 426 \left\{ \begin{array}{l} 426\ 653\ \ldots \\ 612\ 613\ 700\ \ldots \end{array} \right. \end{array} \right. \dashrightarrow 087\ 154\ 170\ 426\ 503 \left\{ \begin{array}{l} 503 \left\{ \begin{array}{l} 503\ 504\ \ldots \\ 908\ 994\ \ldots \end{array} \right. \\ 612 \left\{ \begin{array}{l} 653\ \ldots \\ 612\ 613\ 700\ \ldots \end{array} \right. \end{array} \right. \quad \ldots
$$

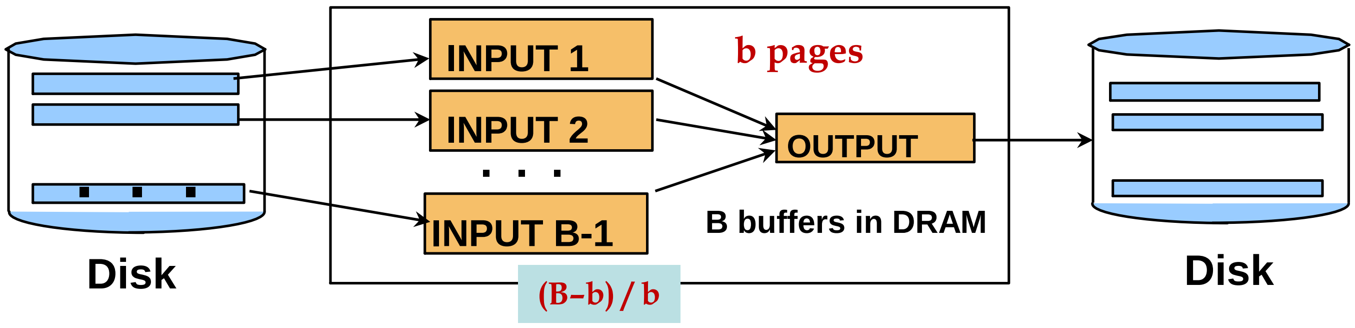
- This optimization reduces # of comparisons down to **log2(B-1)** (if buffer size >> 100, considerable improvement!)

13.3.1 ~ 13.4.2
External Merge-Sort: Some Optimization Techniques

- Replacement sort
  - To minimize # of initial runs by generating longer initial runs,
- Blocked I/O
  - To optimize merge passes
- Double buffering
  - By overlapping I/O, to maximize CPU utilization

External Merge-Sort: Blocked I/O

- To optimize merge pass phase, that is, optimized I/O time
  - Assume B buffers and blocking factor b
    - ✓ all input and output buffer of b pages
    - ✓ can merge at most (B–b) / b runs in each pass



  - Trade-off: read pages in batch VS. more merge passes
  - HOWEVER! current main memory size is large enough ….

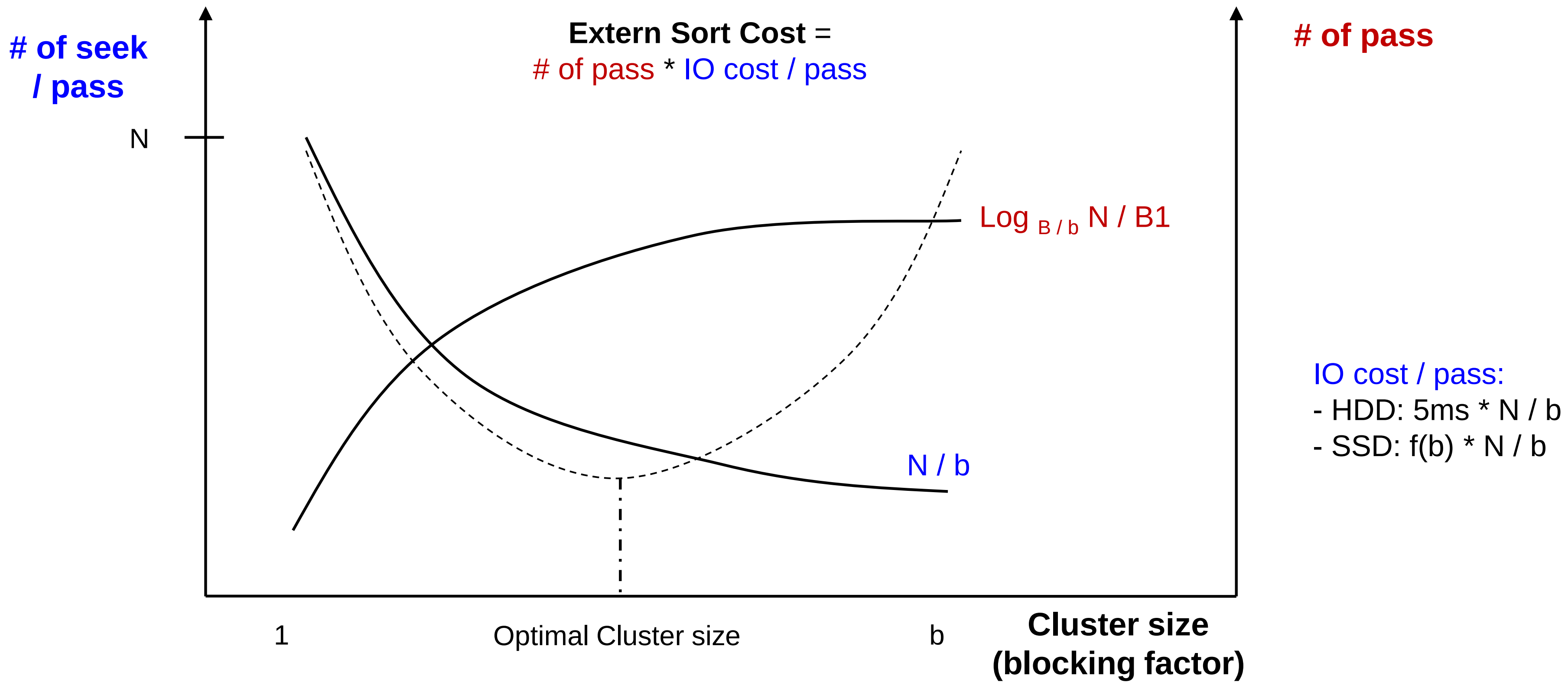
External Merge-Sort: # of Passes in Blocked I/O

N	B=1,000	B=5,000	B=10,000
100	1	1	1
1,000	1	1	1
10,000	2	2	1
100,000	3	2	2
1,000,000	3	2	2
10,000,000	4	3	3
100,000,000	5	3	3
1,000,000,000	5	4	3

* Block size = 32, initial pass produces runs of size 2B.

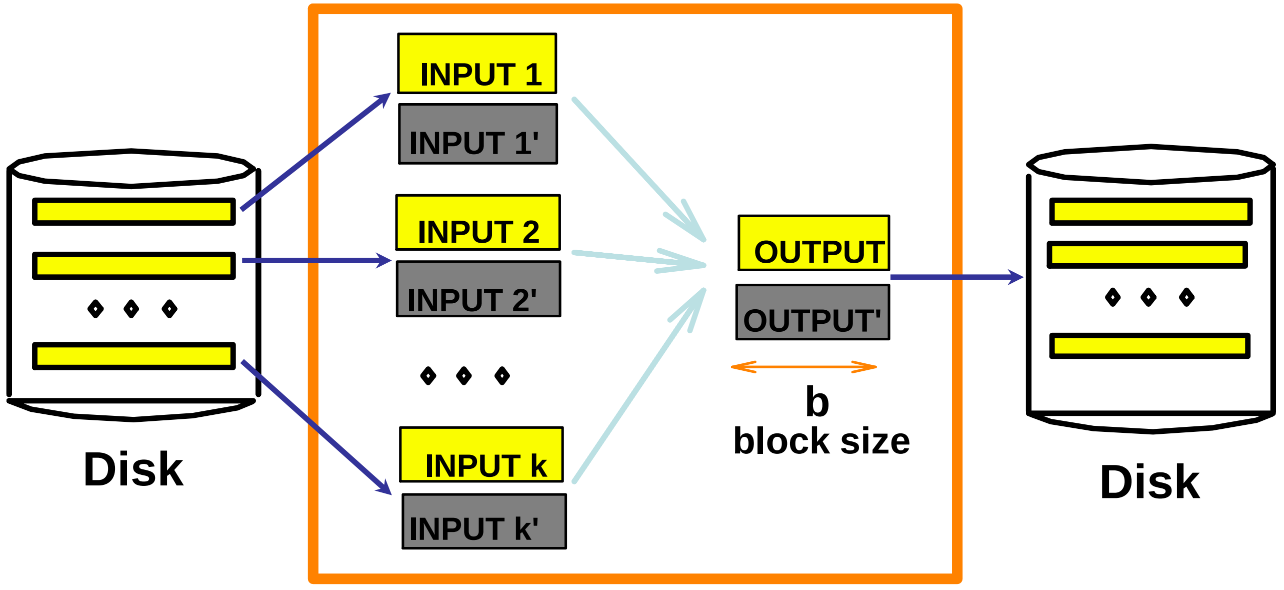
Optimal Cluster Size

- Blocking factor = cluster size
  - Optimal tradeoff: cluster size vs. # of merge pass



External Merge-Sort: Double Buffering

- What happens when an input buffer is consumed?
- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.
- Optimize response time, not throughput



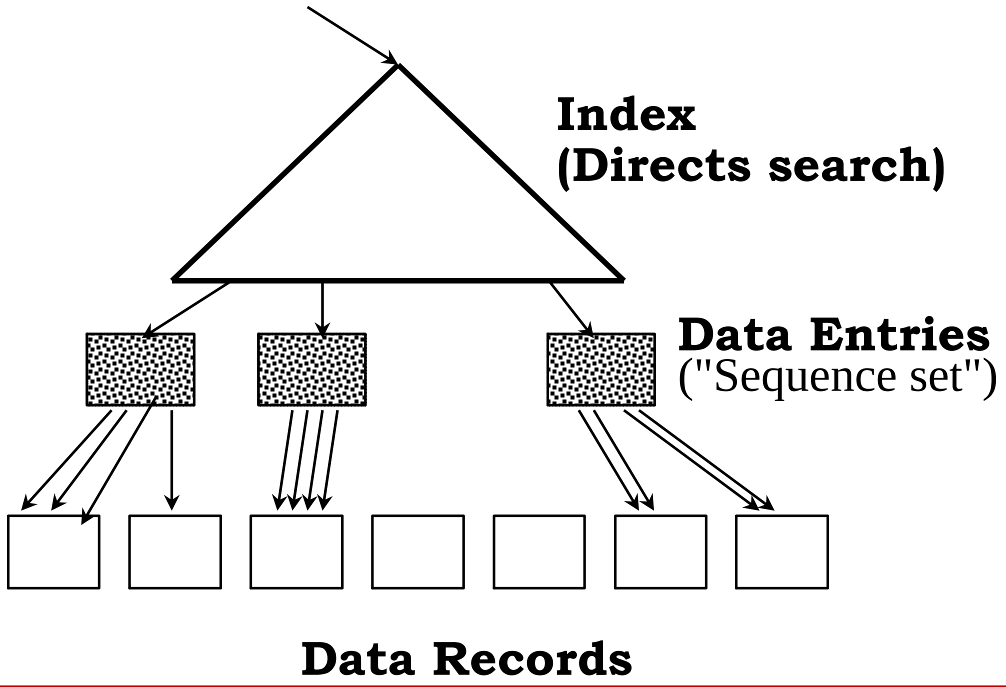
B main memory buffers, k-way merge

13.5 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.
  - good idea??
- Cases to consider:
  - Clustered B+ tree: good idea!!
  - Non-clustered B+ tree: could be a very bad idea!!

Clustered B+ Tree Used for Sorting

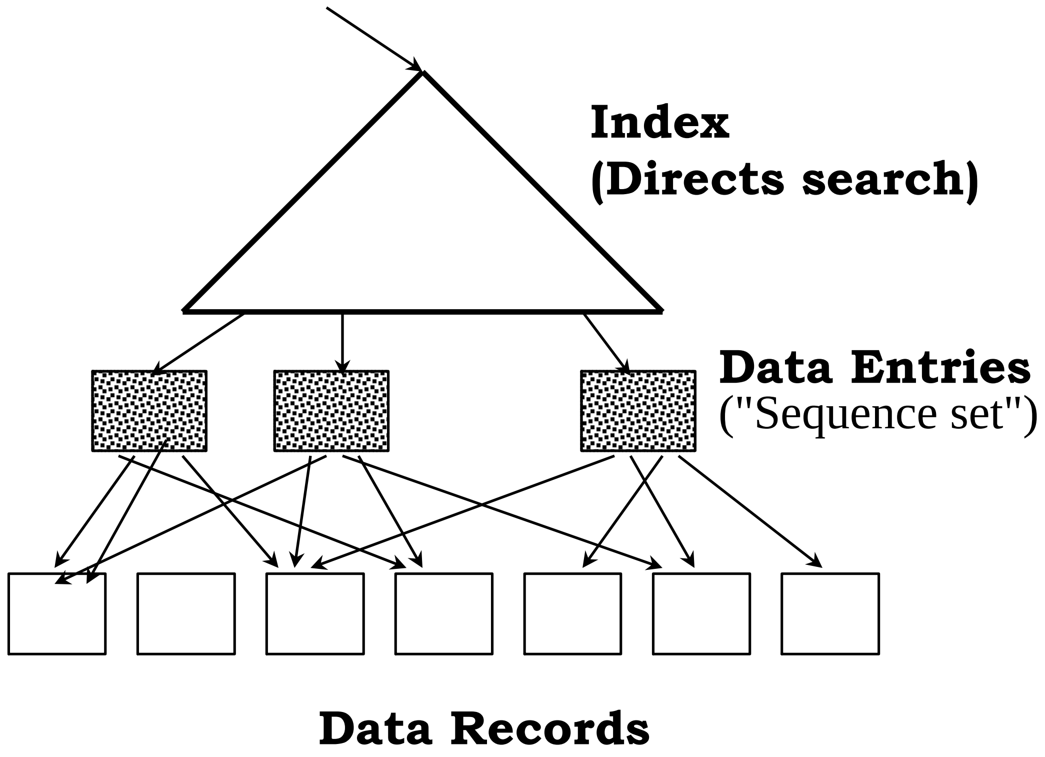
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.



Always better than external sorting!!

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!



Summary

- External sorting is important; DBMS may dedicate part of buffer pool for sorting!

- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# 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 pass.
  - Larger block size means smaller # runs merged.
  - In practice, # of runs (-> passes) rarely more than 2 or 3.

Summary

- Choice of internal sort algorithm may matter:
  - Quicksort: Quick!
  - Heap/tournament sort: slower (2x), longer runs
- The best sorts are wildly fast:
  - Despite 50+ years of research, the sorting algorithms are still improving!
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.