BBM371- Data Management

Lecture 10 External Sorting

20.12.2018

Why Sort in Databases

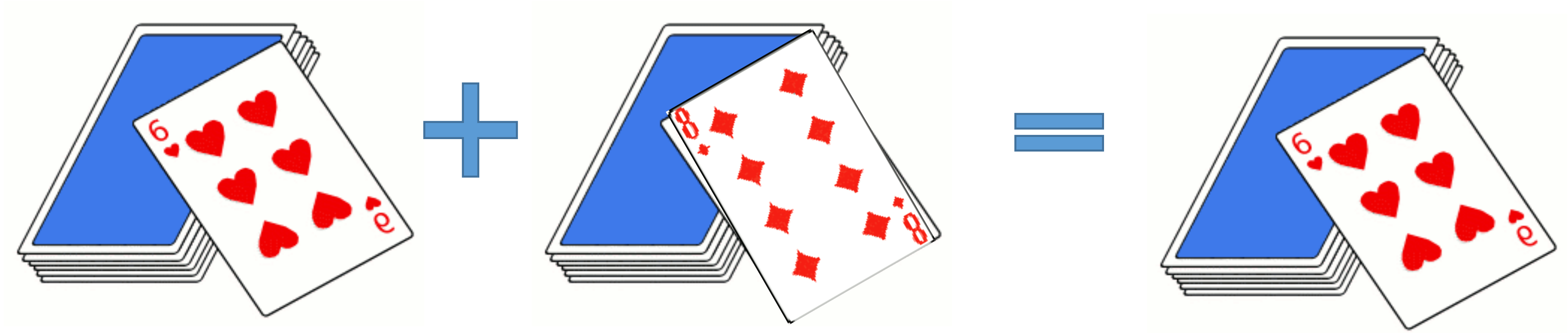
- Data can be requested in sorted order
  - SELECT * FROM Foo ORDER BY bar
- Loading data to an index. Bulk Loading in B+ Tree
- De-duplication (for example DISTINCT keyword in SQL). Remove duplicates by first ordering the records.
- Other query related operations such as joining multiple tables
- So, we will need to sort the keys, records for different needs

In-memory sorting vs. External Sorting

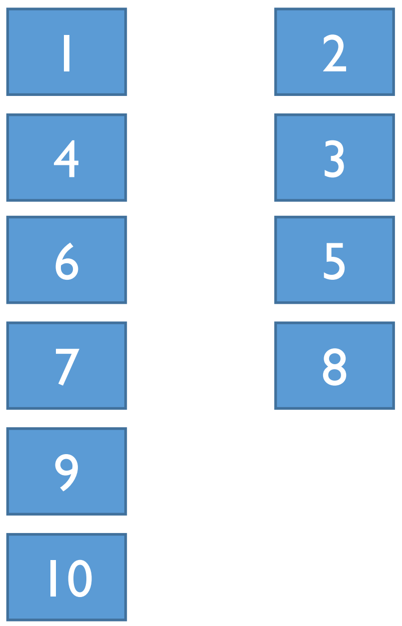
- You have seen sorting algorithms in your previous courses
- Why is it different for databases?
- The data can be much larger than the memory. In this case we have to think about the number of disk accesses.
- Running a sorting algorithm on disk will result in many record swaps that must be performed as disk operations. Will not work!
- We must design an algorithm which uses the limited primary memory wisely, and minimizes the disk accesses.
- Problem: Sort 1GB data with 1MB memory

Merge of Merge Sort

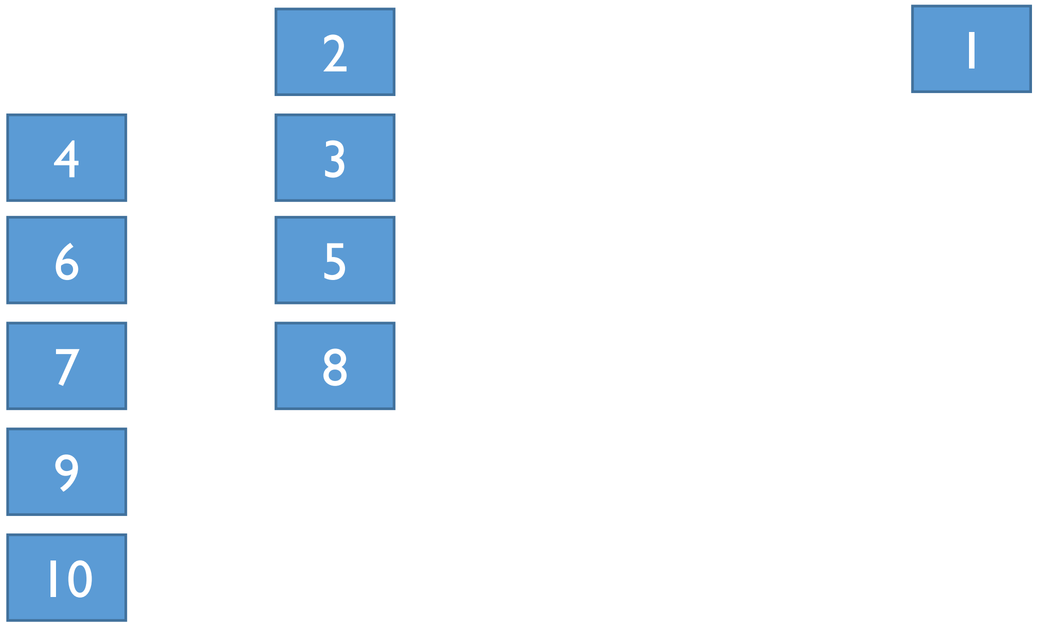
- Merge Sort algorithm is characterized by the merge operation
- Given two sorted lists, merge them to produce a single sorted list
- You can visualize the algorithm as merging two sorted deck of cards.
- Pick the smallest card (on top) from both decks.
- Since we know that the smaller card is not in the other deck, we can add it to the output deck.



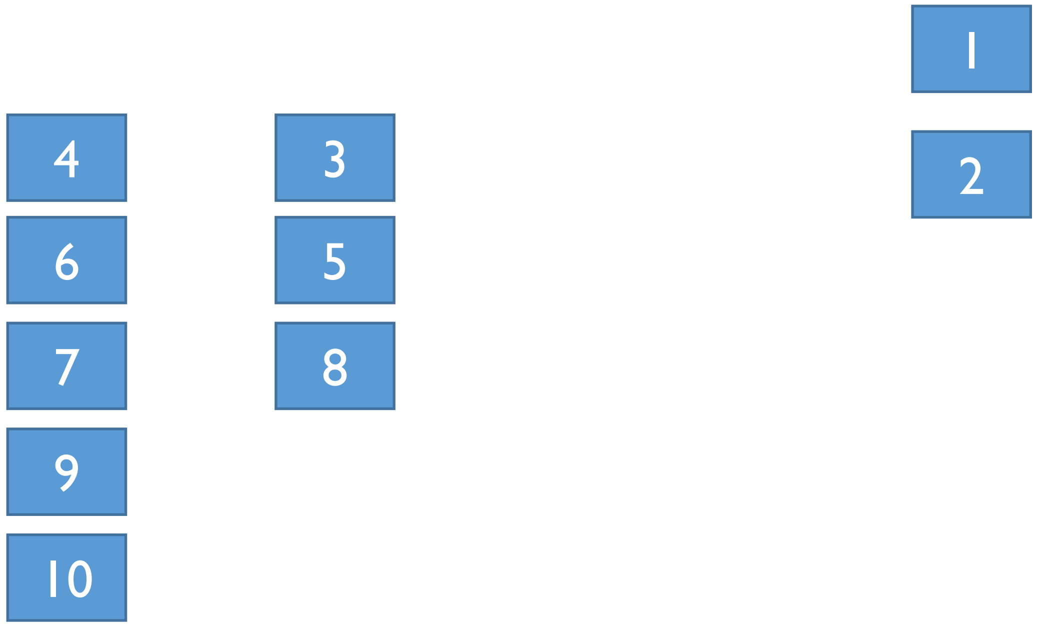
Merge Sort Illustration



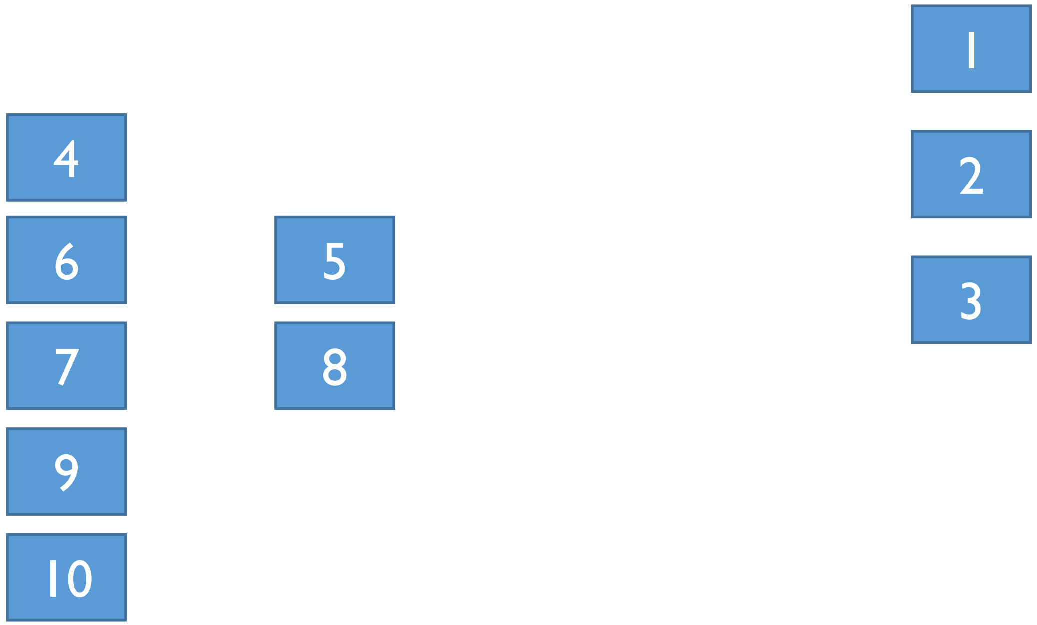
Merge Sort Illustration



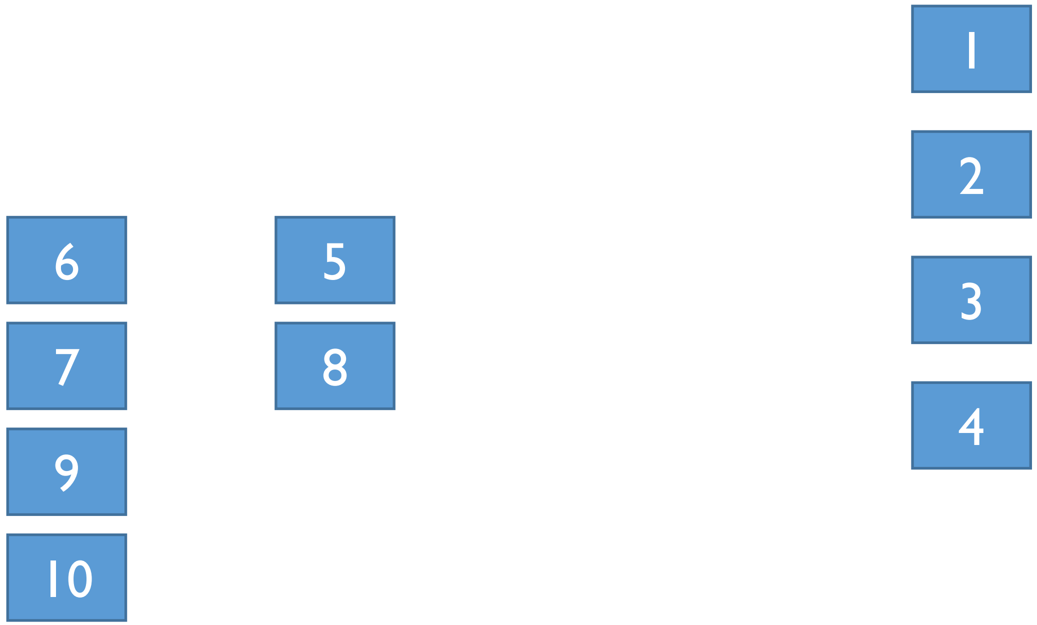
Merge Sort Illustration



Merge Sort Illustration

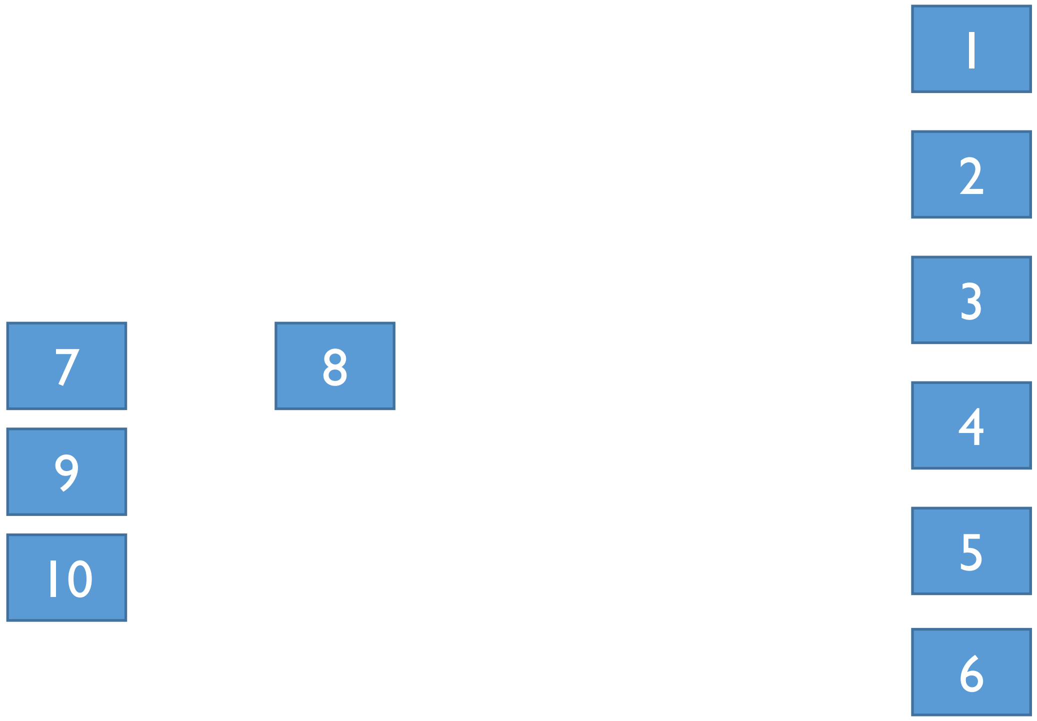


Merge Sort Illustration

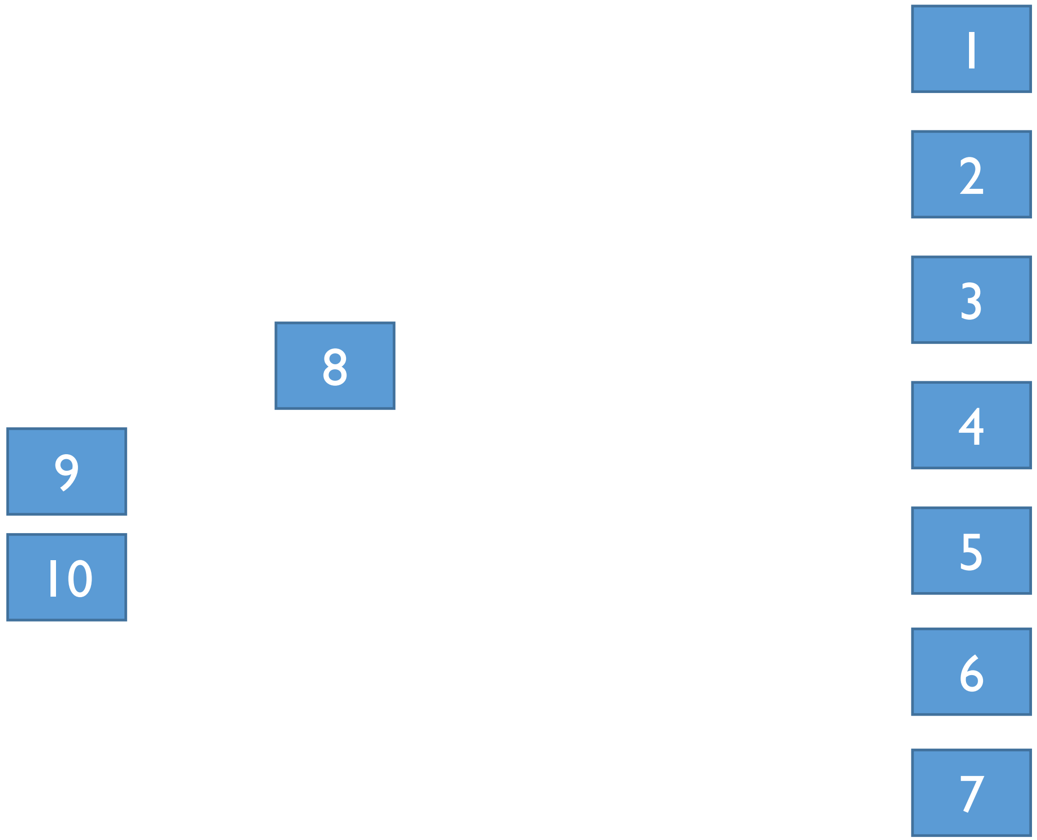


Merge Sort Illustration

Merge Sort Illustration



Merge Sort Illustration



Merge Sort Illustration

1
2
3
4
5
6
7
8

9
10

Merge Sort Illustration

1

2

3

4

5

6

7

8

9

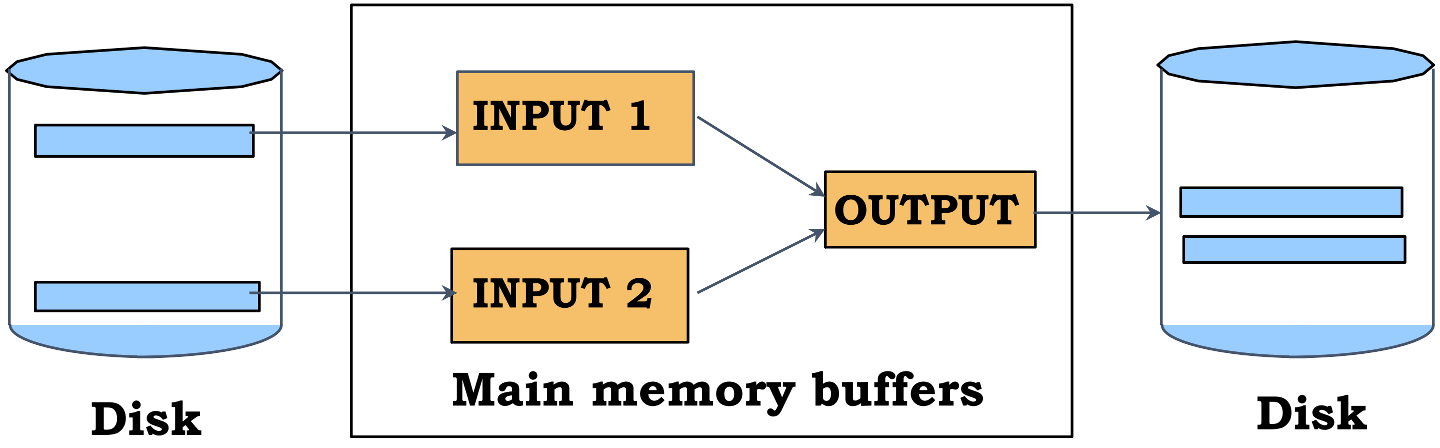
10

General Idea

- Use the memory for sorting a part of the file
  - We will first consider sorting a page in memory
- Use the merge operation to merge runs until the whole file is sorted!
- The merge can be implemented for any two sublists of arbitrarily large sizes, as we only need the top of the decks

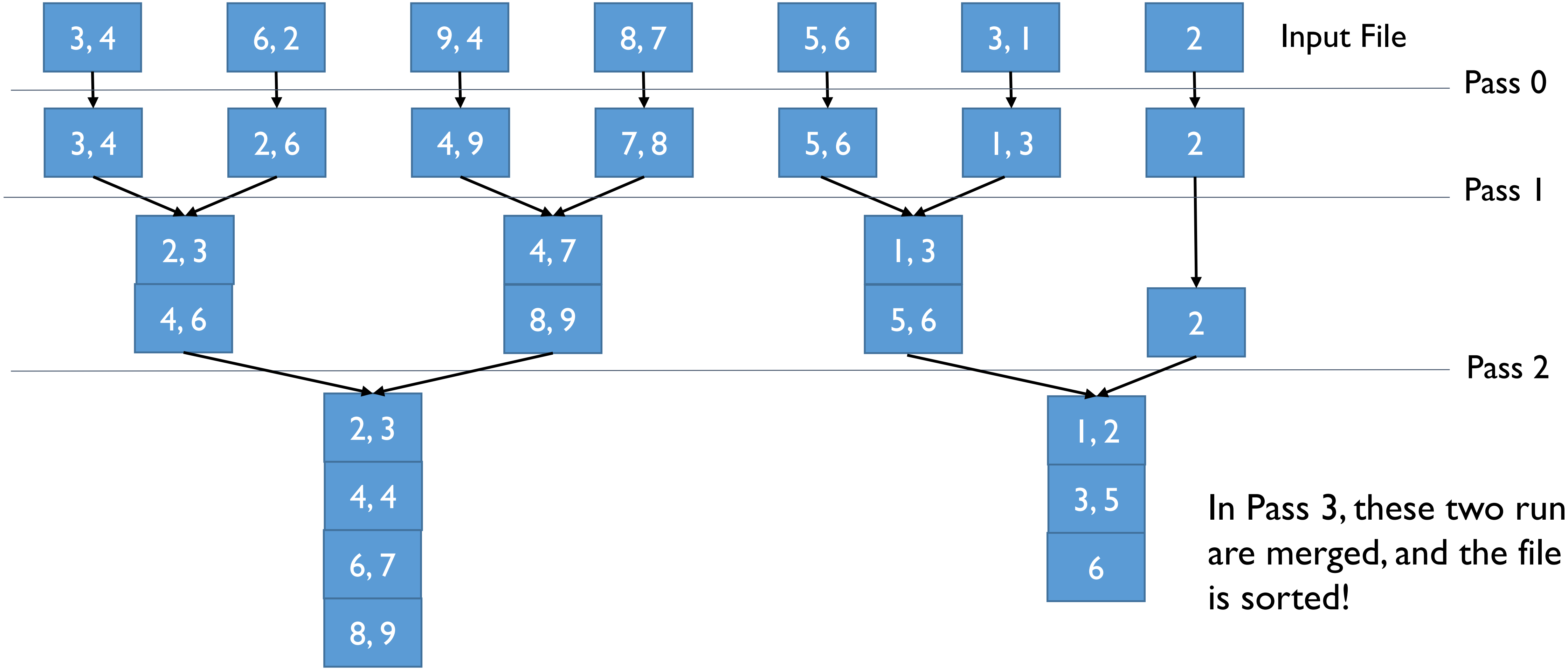
2-Way Sort

- Read a page, sort it, write it
  - Only one buffer page is used
  - Written sorted page is called as a run
- Pass 2, 3 … etc:
  - 3 page memory buffer: 2 for reading runs, Merge & Write in one page



2-Way Merge Example

- Assume that we have 7 pages in disk.
- Each page can store 2 keys



In Pass 3, these two runs are merged, and the file is sorted!

2-Way Merge Sort Algorithm

- proc 2-way-extsort (file)
  - Read each page into memory, sort it, write it out //Produce runs that are one page long – Pass 0

    //Pass i=1,2, …
  - While the number of runs at end of previous pass is > 1
    - While there are runs to be merged from previous pass:
      - Choose next two runs (from previous runs)
      - Read each run into an input buffer; page at a time
      - Merge the runs and write to the output buffer
      - Force output buffer to disk one page at a time

endproc

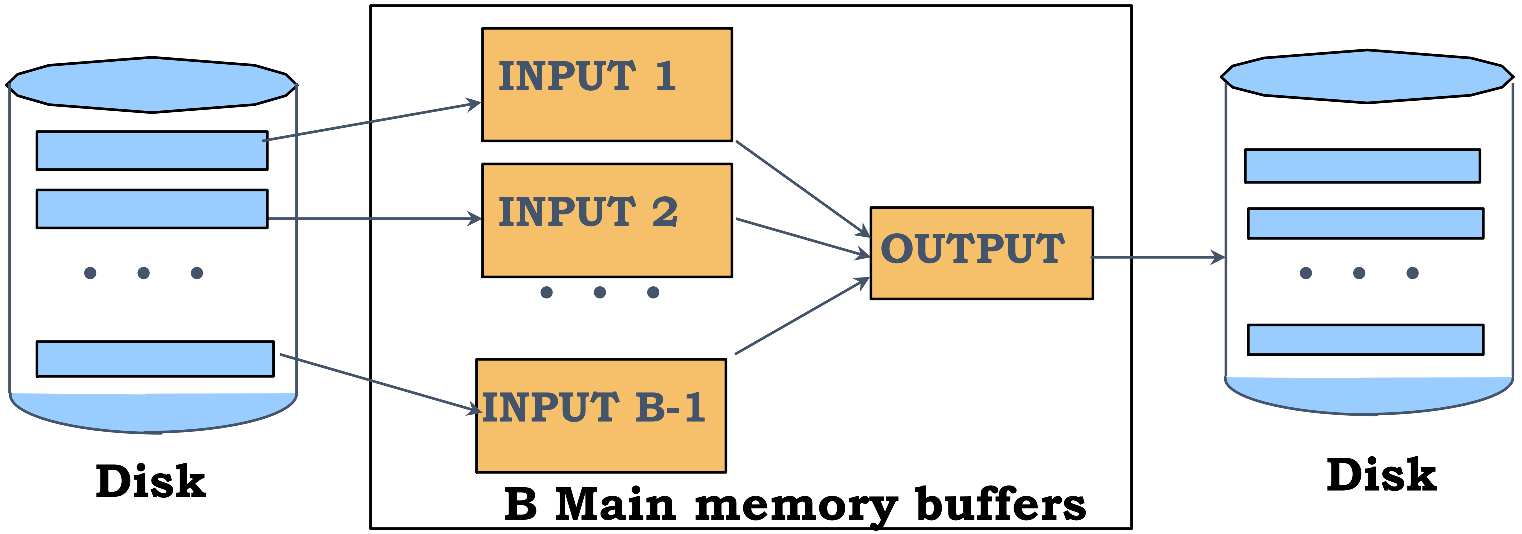
Requires just three buffer pages!

2-Way Merge Analysis

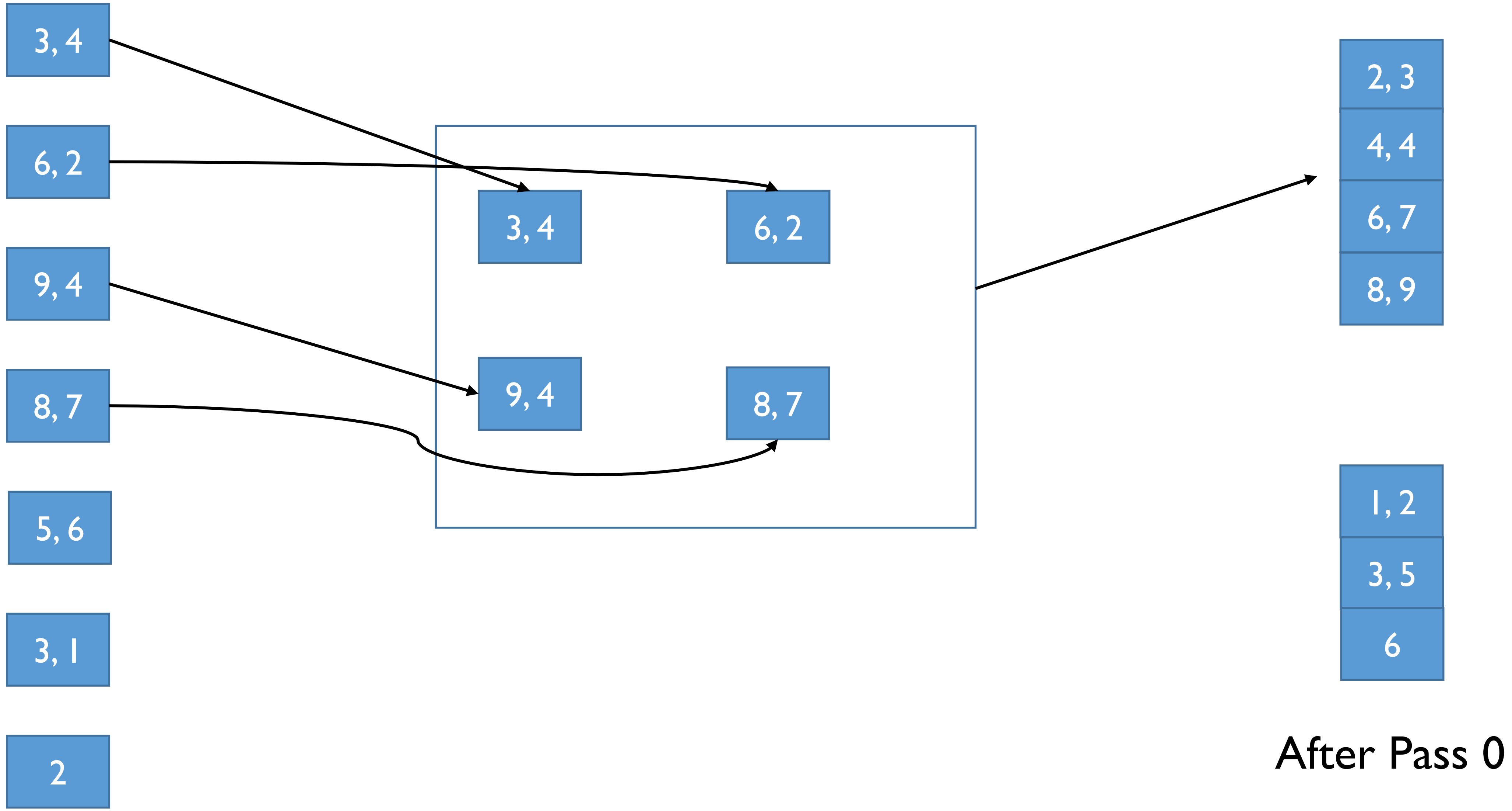
- The number of passes needed for sorting a file of 2^k is k
- At each step we are merging two runs. So ceil($\log_2$ N), where N is the number of pages in the file. If we add the initial Pass, ceil($\log_2$N) + 1
- In each pass we have to read and write each page. So, the cost for a pass is 2N
- The total cost of this procedure is 2N* ceil($\log_2$N + 1)
- So for our example, with 7 pages. We have 4 passes. At each pass we have 2*7 disk access. The total cost is 56 disk accesses.

How to do better?

- The complexity increases as the number of passes increases.
- Instead of merging two runs at a time, we will merge as much runs as it is possible
  - Number of runs that can fit in the memory
- Instead of 2 we use B pages to read to memory (e.g. B=4 pages)
  - Pass 0 : Create ceil(N/B) runs (N: number of pages in the file)
  - Pass 1…k: Merge B-1 runs (1 page is used for the output)

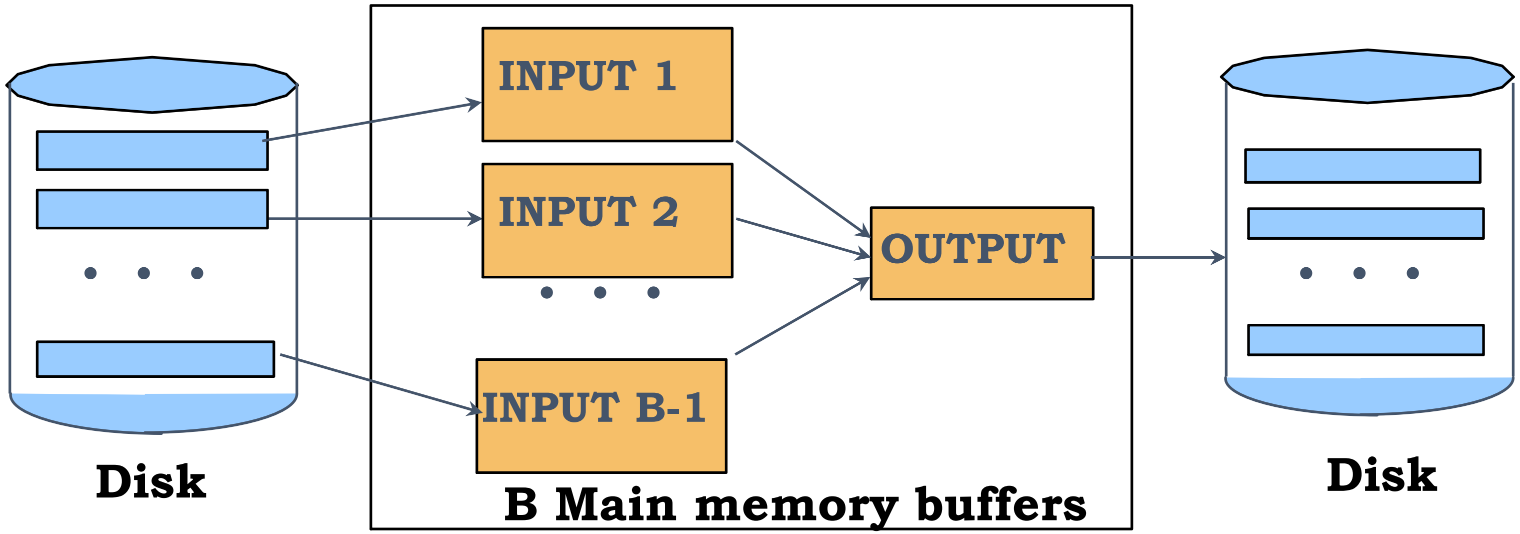


First Sort (Pass 0) B=4



B-1 Way Merge

- After this step, we can merge B-1=3 different runs at the same time
- Algorithm for B-1 Way Merge is similar to two-way case:
  - At each step choose the smallest key
  - Write to output



Cost of External Merge Sort

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

A note on complexity

- The asymptotic complexity of both algorithms is O(N log N)
- The base of logarithm can be changed by the rule

$\log_a b = \log_c b / \log_c a$

So a different base changes only the constant of the cost!

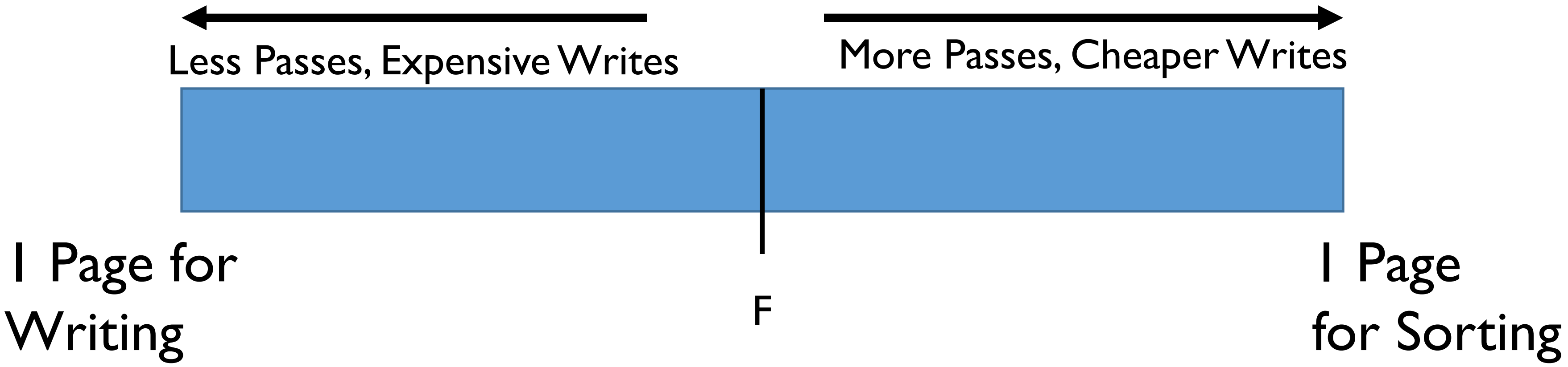
- However, when the base is (B-1) the number of passes will drastically decrease

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

Cost of Disk Operations

- We have seen that performing a disk write or read for a continous sequence of files is more efficient.
- So, for example writing the whole file a page at a time will take more time than writing F pages at a time.
- Use B-F pages for sorting, reserve F blocks for writing.



Sorting Records!

- Sorting has become a blood sport!
  - Parallel sorting is the name of the game ...
- Datamation: Sort 1M records of size 100 bytes
  - Typical DBMS: 5 minutes
  - 2001: .48sec at UW
- New benchmarks proposed:
  - Minute Sort: How many TB can your sort in 1 minute?
    - 2016: 37TB/55TB Tencent Sort at Tencent Corp., China
  - Cloud Sort: How much in $ to sort 100 TB using a public cloud?
    - 2015: $451 on 330 Amazon EC2 r3.4xlarge nodes, by profs at UCSD.
    - 2016: $144 using Alibaba Cloud, by profs at Nanjing U, others

Example

- Assuming that our most general external sorting algorithm is used. For a file with 2,000,000 pages and 17 available buffer pages, answer the following

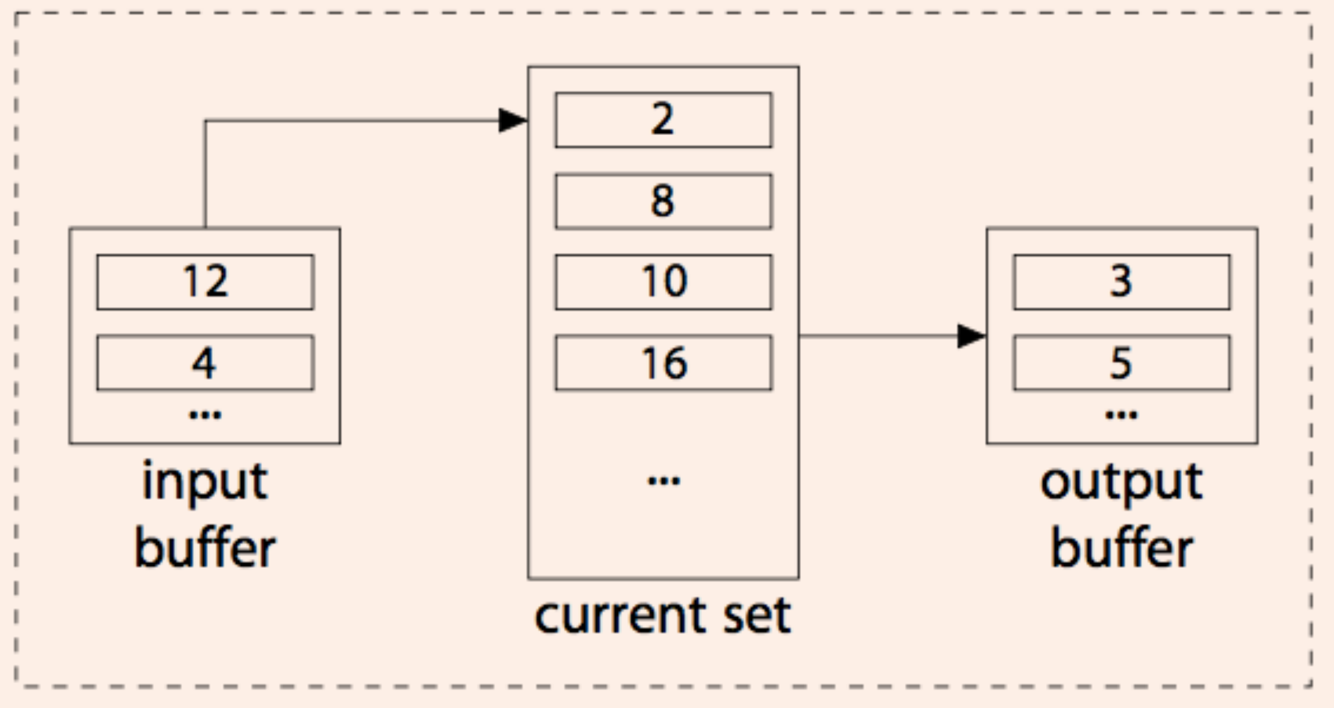
1. How many runs will you produce in the first pass?
2. How many passes will it take to sort the file completely?
3. What is the total I/O cost of sorting the file?
4. How many buffer pages do you need to sort the file completely in just two passes?

Answer

- How many runs are produced in Pass 0
  - Ceil(2000000/17) = 117648 sorted runs.
- Number of passes required
  - *Ceil*(log_{16}117648) + 1 = 6 passes.
- Total number of disk accesses
  - 2*2000000*6 = 24000000.
- How many Buffer pages do we need, to complete sort in two passes
  - We have to produce less than equal to B-1 runs after first pass. So ceil(N/B) must be less than equal to B-1. For $2*10^6$ pages, if $B=10^3$ we have 2000 runs, B=1415 produces 1414 runs which can be merged in single pass.

Replacement Sort

- **Replacement sort** can help to further cut down the number of initial runs $\lceil N/B \rceil$: try to **produce initial runs with more than** B **pages**.
- Assume a buffer of B pages. Two pages are dedicated **input** and **output buffers**. The remaining $B - 2$ pages are called the **current set**:



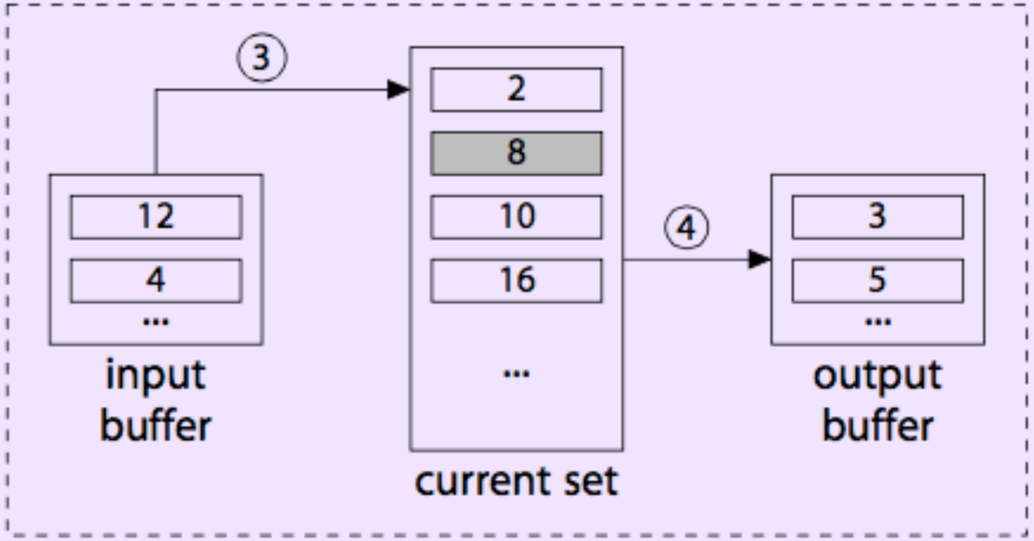
Replacement Sort Algorithm

Replacement sort

1. Open an empty run file for writing.
2. Load next page of file to be sorted into input buffer. If input file is exhausted, go to 4.
3. While there is space in the current set, move a record from input buffer to current set (if the input buffer is empty, reload it at 2).
4. In current set, pick record r with smallest key value k such that $k \geqslant k_{out}$ where k_{out} is the maximum key value in output buffer.[1] Move r to output buffer. If output buffer is full, append it to current run.
5. If all k in current set are $< k_{out}$, append output buffer to current run, close current run. Open new empty run file for writing.
6. If input file is exhausted, stop. Otherwise go to 3.

Replacement Sort Example

Example (Record with key $k = 8$ will be the next to be moved into the output buffer; current $k_{out} = 5$)



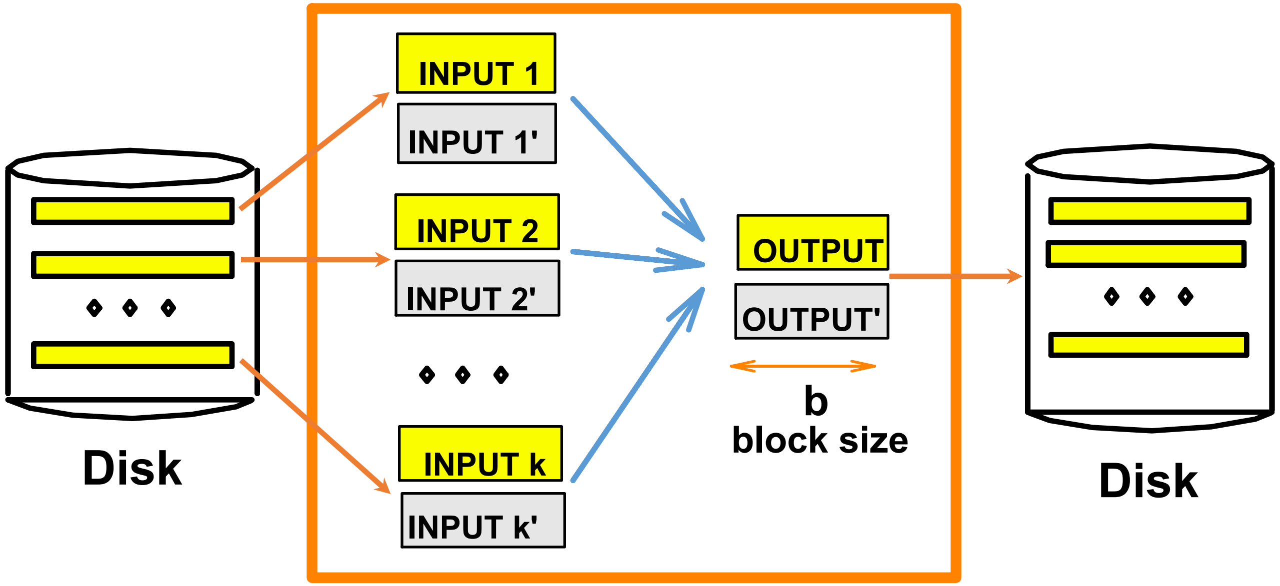
- The record with key $k = 2$ remains in the current set and will be written to the subsequent run.

External Sort
Remarks

- External sorting follows a **divide and conquer** principle.
  - This results in a number of **independent (sub-)tasks.**
  - **Execute tasks in parallel** in a distributed DBMS or exploit multi-core parallelism on modern CPUs.
- To keep the CPU busy while the input buffer is reloaded (or the output buffer appended to the current run), use **double buffering**.

Double Buffering

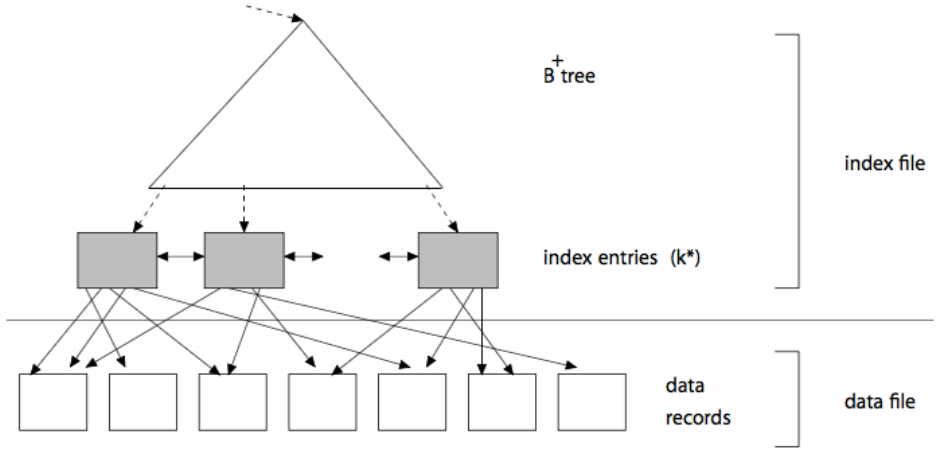
- While waiting to read or write a block, the CPU will be idle as we have no data to process.
- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.
  - Potentially, more passes; in practice, most files *still* sorted in 2-3 passes.



B main memory buffers, k-way merge

B+ Trees for Sorting

- If a B+-tree matches a sorting task (*i.e.*, B+-tree organized over key *k* with ordering θ), we *may* be better off to **access the index and abandon external sorting**.
- 1) If the B+-tree is **clustered**, then
  - the data file itself is already θ-sorted,
  - simply sequentially read the sequence set (or the pages of the data file).



- 2) If the B+-tree is **unclustered**, then
  - in the worst case, we have to initiate one I/O operation per record (not per page)! ⇒ do not consider the index.

Summary

- External sorting is important; DBMS may dedicate part of buffer pool just for sorting!
- External merge sort minimizes disk I/O cost.
- Using clustered B+ tree for sorting is always better than external sorting. Root to the leftmost leaf, then retrieve all leaf pages.
- Using an unclustered B+ tree for sorting could be a very bad idea.
- The best sorts are already fast but still we're still improving!