Universität
Konstanz

Database System Architecture and Implementation

Module 6
External Sorting
December 3, 2018

Orientation

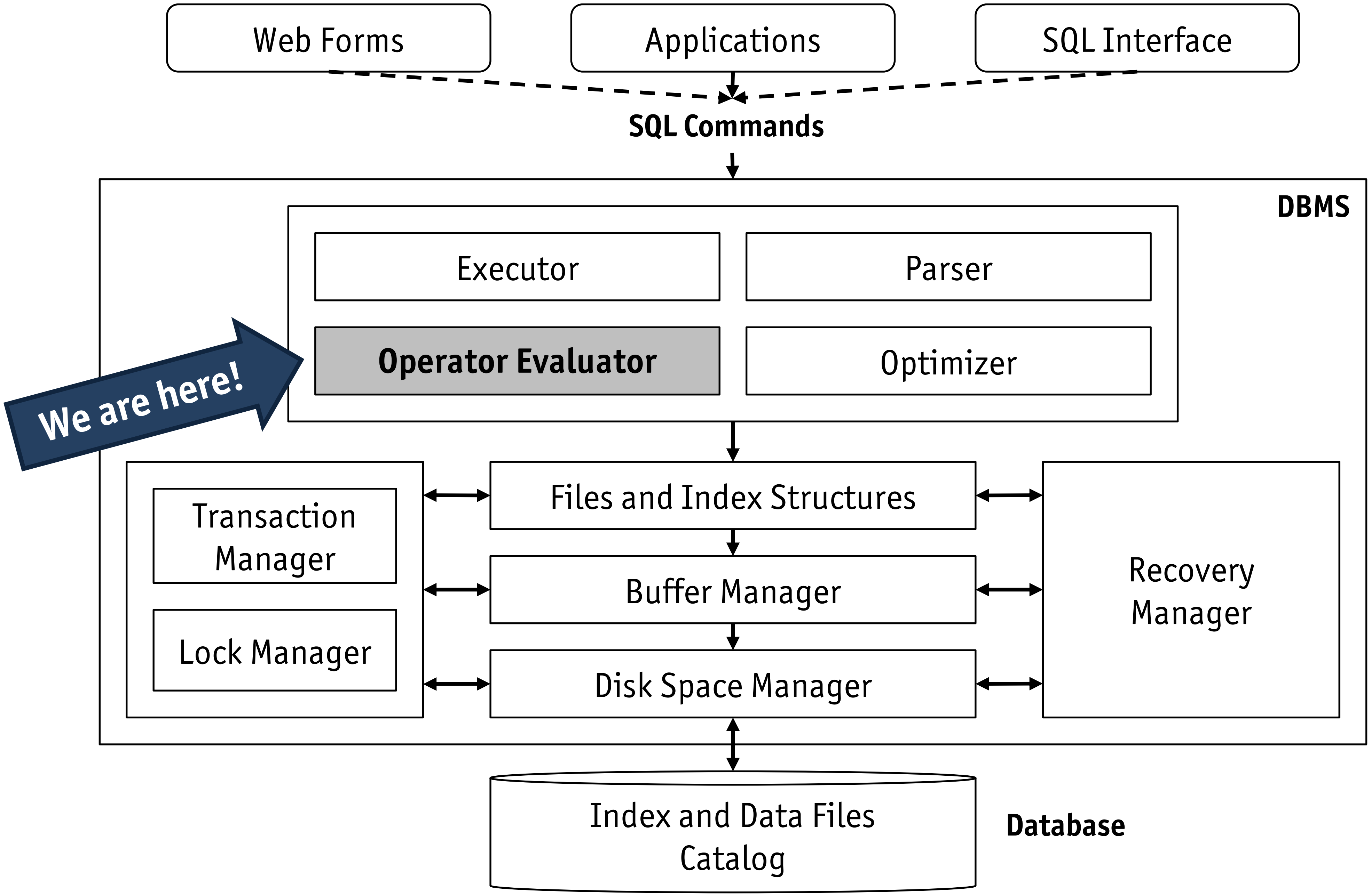


Figure Credit: Raghu Ramakrishnan and Johannes Gehrke: "Database Management Systems", McGraw-Hill, 2003.

Module Overview

- Overview of sorting
- Two-way merge sort
- External merge sort
  - longer initial runs using selection sort
  - better CPU usage using double buffering
- Using B+ trees for sorting

Sorting

A (not so) simple SQL query

```
SELECT DISTINCT S.sid, S.sname, S.rating
  FROM Reserves AS R, Sailors AS S
 WHERE R.sid = S.sid
 GROUP BY S.sid, S.sname, S.rating, R.day
HAVING COUNT(R.bid) > 1
 ORDER BY S.rating DESC
```

Possible reasons to sort

Which operations in the above query might require sorting?

Sorting

Definitions

- A file is **sorted** with respect to **sort key** k and **ordering** θ, if for any two records r_1, r_2 in the file, their corresponding keys are in θ-order

$$r_1 \, \theta \, r_2 \quad \Leftrightarrow \quad r_1.k \, \theta \, r_2.k$$

- A key may be a single attribute as well as an ordered list of attributes. In the latter case, order is defined **lexicographically**. Consider k = ($\mathtt{A}$, $\mathtt{B}$), θ = <

$$r_1 < r_2 \quad \Leftrightarrow \quad r_1.\mathtt{A} < r_2.\mathtt{A} \lor (r_1.\mathtt{A} = r_2.\mathtt{A} \land r_1.\mathtt{B} < r_2.\mathtt{B})$$

Sorting

- If the data to be sorted is too large to fit into available main memory (buffer pool), an **external sorting** algorithm is required
- Stepwise design of an external sorting algorithm
  1. **simple algorithm:** only three pages of buffer space are sufficient to sort a file of arbitrary size
  2. **refined algorithm:** simple algorithm can be adapted to make effective use of larger (and more realistic) buffer sizes
  3. **optimized algorithm:** a number of further optimizations can be applied to reduce the number and duration of required page I/O operations

Two-Way Merge Sort

- Two-way merge sort can sort files of arbitrary size with only **three pages** of available buffer space

Two-way merge sort

Two-way merge sort sorts a file with $N = 2^k$ pages in multiple **passes**, each of which produces a certain number of sorted sub-files, so-called **runs.**

- **Pass 0** sorts each of the 2^k input pages individually and in **main memory**, resulting in 2^k sorted runs.
- **Subsequent passes** merge pairs of runs into larger runs. Pass n produces 2^{k-n} runs.
- **Pass** k produces only one final run, the overall sorted result.

During each pass, every page of the file is read and written. Therefore, a total of $\mathbf{2 \cdot k \cdot N}$ **page I/O operations** are required to sort the file.

Exercise: forwards in Pass 0?

What happens if the pages that are sorted in Pass 0 contain **forwards** to moved records?

Two-Way Merge Sort

Pass 0 (**input:** $N = 2^k$ unsorted pages, **output:** 2^k sorted pages)

1. **Read** N pages, one page at a time
2. **Sort** records, page-wise, in main memory
3. **Write** sorted pages to disk (each page results in a **run**)

This pass requires **one page** of buffer space

Pass 1 (**input:** $N = 2^k$ sorted pages, **output:** 2^{k-1} sorted runs)

1. **Open** two runs r_1 and r_2 from Pass 0 for reading
2. **Merge** records from r_1 and r_2, reading input page by page
3. **Write** new two-page run to disk, page by page

This pass requires **three pages** of buffer space

⋮

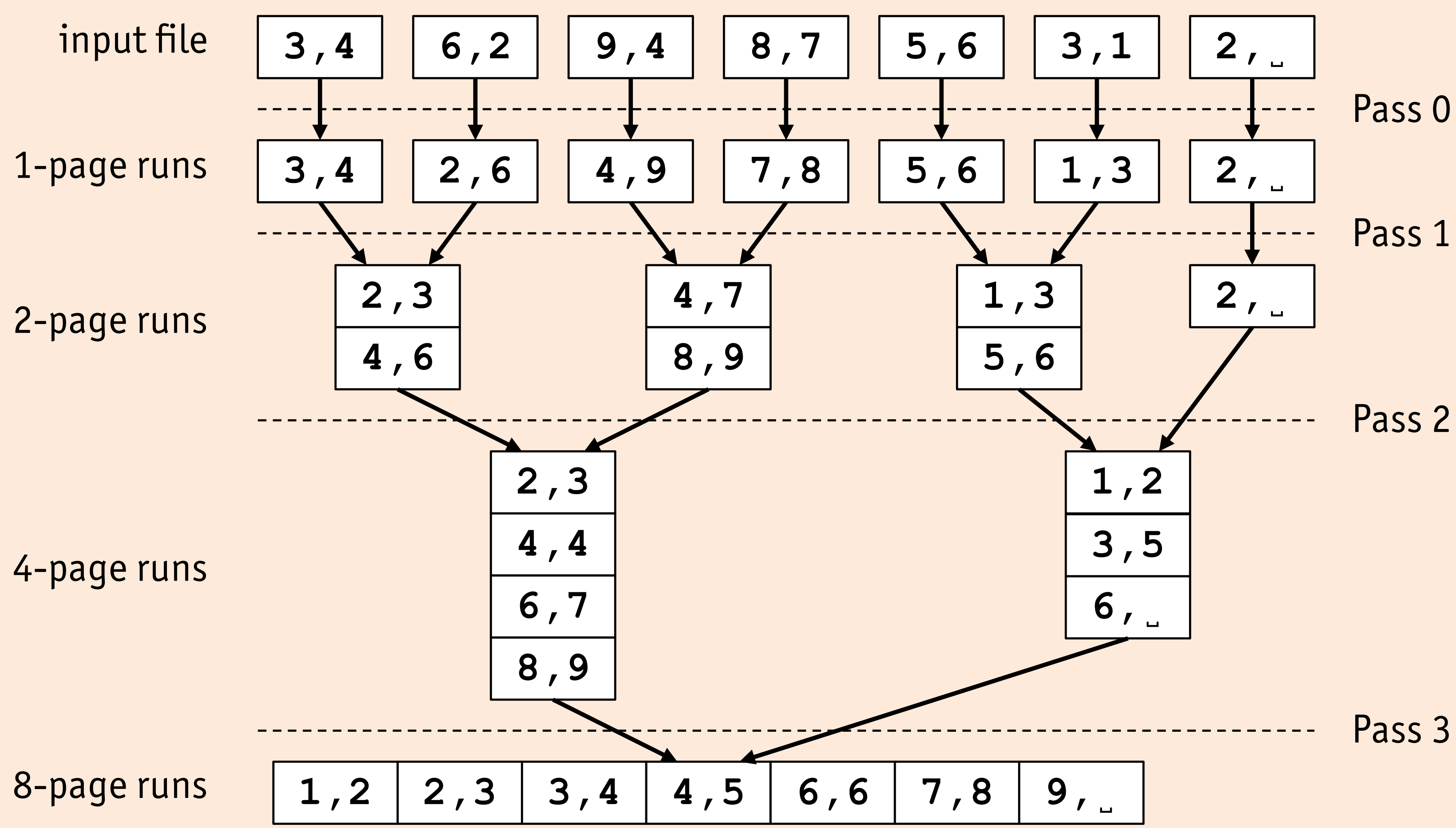
Pass n (**input:** $N = 2^{k-n+1}$ sorted runs, **output:** 2^{k-n} sorted runs)

1. **Open** two runs r_1 and r_2 from Pass $n - 1$ for reading
2. **Merge** records from r_1 and r_2, reading input page by page
3. **Write** new 2^n-page run to disk, page by page

This pass requires **three pages** of buffer space

Two-Way Merge Sort

Example: 7-page file, 2 records/page, keys *k* shown, θ = <



Two-Way Merge Sort

Two-way merge sort ($N = 2^k$, ordering θ, output in file "run␣k␣0")

```
function 2-way-merge-sort (file, N)
    for page number p in 0...2^k – 1 do          (Pass 0: write 2^k sorted single-page runs)
        ↑p ← pin (file, p) ;
        f_0 ← createFile ("run␣0␣r") ;          ("run␣n␣r" contains the r^th run of Pass n)
        sort the records on page pointed to by ↑p according to θ ;
        write page pointed to by ↑p into file f_0 ;
        closeFile (f_0) ;
        unpin (file, p, false) ;
    for n in 1...k do                             (Passes 1...k)
end
```

- Remark
  - the in-memory sort and merge steps can be implemented using **standard sorting techniques**, e.g., quick-sort

Two-Way Merge Sort

Two-way merge sort ($N = 2^k$, ordering θ, output in file "run␣k␣0")

function `2-way-merge-sort` (*file*, N)
  for each page number p in $0 \ldots 2^k - 1$ **do** (Pass 0: write 2^k sorted single-page runs)
  for n in $1 \ldots k$ **do** (Passes $1 \ldots k$)
    for r in $0 \ldots 2^{k-n}$ **do** (pair-wise merge all runs written in Pass $n - 1$)
      $f_1 \leftarrow$ **openFile** ("run␣$(n-1)$␣$(2 \cdot r)$") ;
      $f_2 \leftarrow$ **openFile** ("run␣$(n-1)$␣$(2 \cdot r + 1)$") ;
      $f_0 \leftarrow$ **createFile** ("run␣n␣r") ;
      for page number p in $0 \ldots 2^{n-1} - 1$ **do**
        $\uparrow p_1 \leftarrow$ **pin** (f_1, p) ; $\uparrow p_2 \leftarrow$ **pin** (f_2, p) ;
        merge the records on pages pointed to by $\uparrow p_1$, $\uparrow p_2$ according to θ ;
        append resulting two pages to file f_0 ; $(\text{size}(f_0) = \text{size}(f_1) + \text{size}(f_2))$
        unpin (f_1, p, false) ; **unpin** (f_2, p, false) ;
      closeFile (f_0) ;
      deleteFile (f_1) ;
      deleteFile (f_2) ;
end

Two-Way Merge Sort

- Each pass in a two-way merge of a file of *N* pages
  - pass **reads** *N* pages in
  - sorts/merges in memory
  - **writes** *N* pages out again

  $\Big\}$ $2 \cdot N$ page I/O operations per pass
- Number of passes

$$\underbrace{1}_{\text{Pass 0}} + \underbrace{\lceil \log_2 N \rceil}_{\text{Passes 1,...,k}}$$

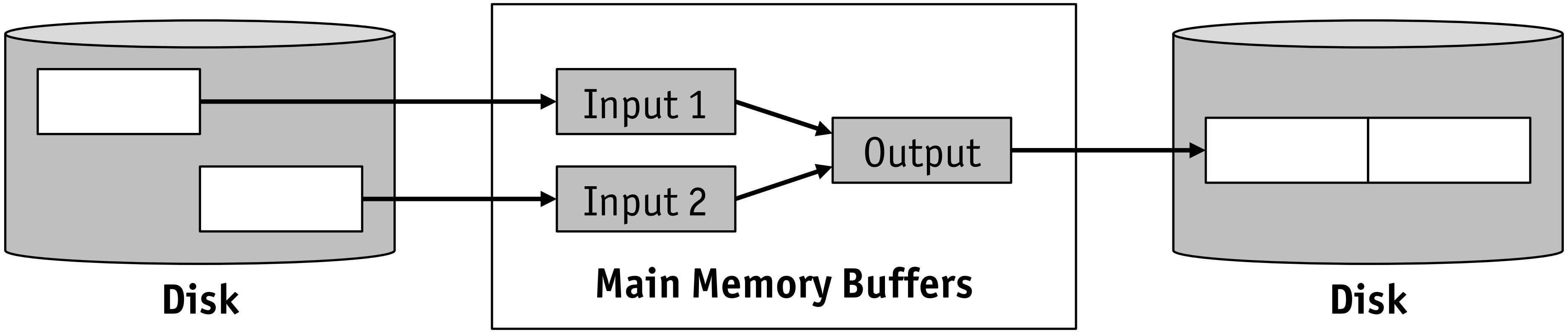
- Total number of I/O operations

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

✎ Exercise: how many page I/O operations does it take to sort an 8 GB file?

Assume a page size of 8 kB (with 1000 records each).

Two-Way Merge Sort

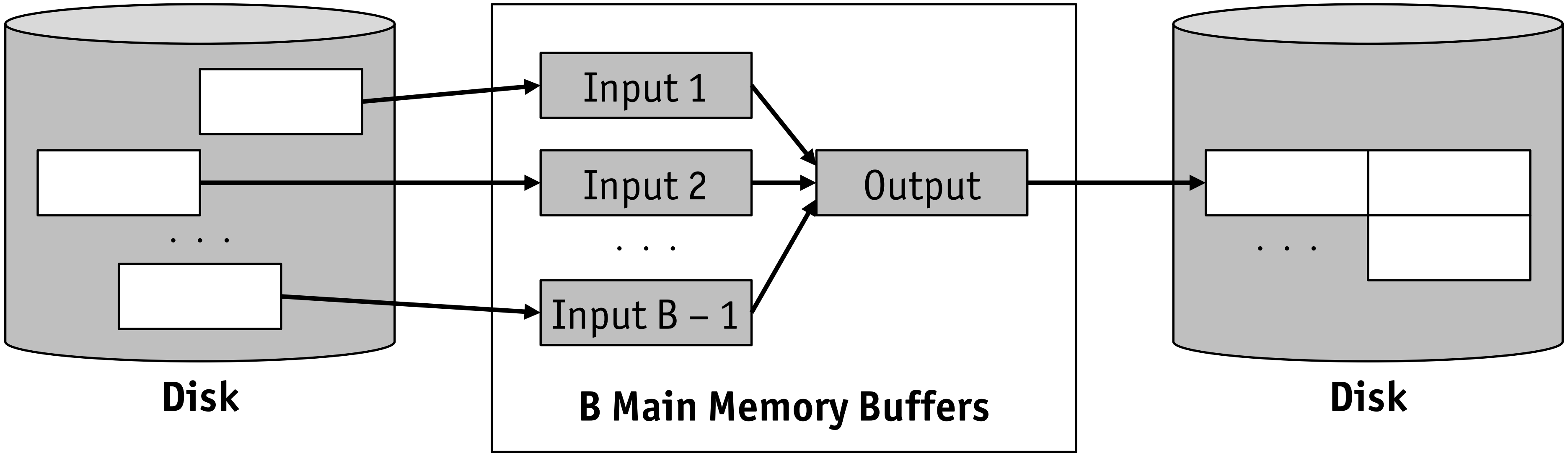


- At any point in time, two-way merge sort uses **no more than three pages** of buffer space
  - consider the **`pin(·)`** calls in the pseudo-code of the algorithm
  - this restriction in "voluntarily"
- In reality, **many** more free buffer pages will be available and this sort algorithm can be refined to make efficient use of them

External Merge Sort

- **External merge sort** introduces two improvements over simple two-way merge sort
  - **reduce the number of runs** by using the full buffer space during to avoid creating one-page runs in Pass 0
  - **reduce the number of passes** by merging more than two runs at a time

External Merge Sort



- Suppose *B* pages are available in the buffer pool
  - *B* **pages can be read at a time** during Pass 0 and sorted in memory
  - *B* – 1 **pages can be merged at a time** (leaving one page as a write buffer)

External Merge Sort

Pass 0 (**input:** N = unsorted pages, **output:** $\lceil N/B \rceil$ sorted pages)

1. **Read** N pages, B pages at a time
2. **Sort** records, page-wise, in main memory
3. **Write** sorted pages to disk (resulting in $\lceil N/B \rceil$ **runs**)

This pass uses B **pages** of buffer space

⋮

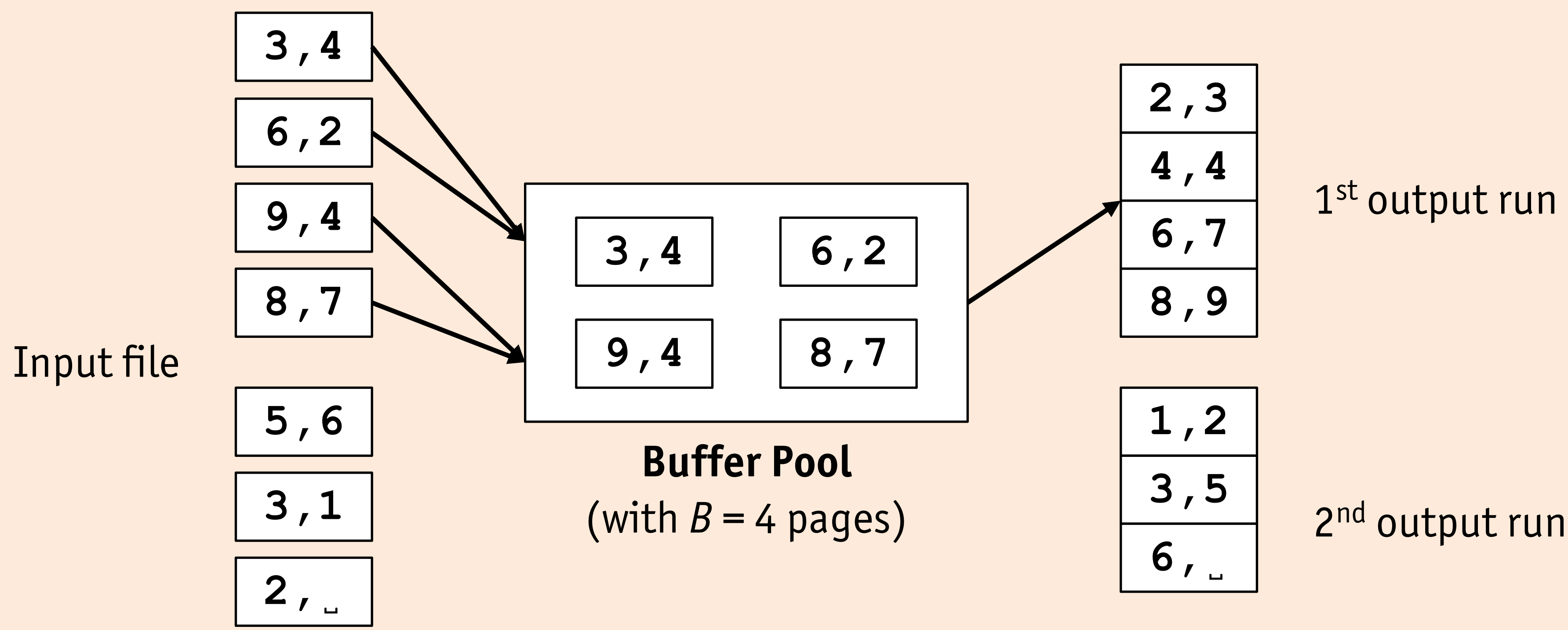
Pass n (**input:** $\frac{\lceil N/B \rceil}{(B-1)^{n-1}}$ sorted runs, **output:** $\frac{\lceil N/B \rceil}{(B-1)^{n}}$ sorted runs)

1. **Open** $B - 1$ runs $r_1, \ldots, r_{B-1}$ from Pass $n - 1$ for reading
2. **Merge** records from $r_1, \ldots, r_{B-1}$, reading input page by page
3. **Write** new $B \cdot (B - 1)^n$-page run to disk, page by page

This pass requires B **pages** of buffer space

External Merge Sort

Example: 7-page file, 2 records/page, keys *k* shown, θ = <, 4 buffer pages, pass 0



External Merge Sort

- As in two-way merge sort, each pass reads, processes, and then writes **all** N pages
- The number of initial runs **determines** the number of passes
  - in Pass 0, $\lceil N/B \rceil$ runs are written
  - number of additional passes is thus $\lceil \log_2 \lceil N/B \rceil \rceil$
- With B pages of buffer space, we can do a $(B - 1)$-**way merge**
  - number of additional passes is thus $\lceil \log_{B-1} \lceil N/B \rceil \rceil$
- Total number of page I/O operations

$$2 \cdot N \cdot \left(1 + \left\lceil \log_{B-1} \left\lceil N/B \right\rceil \right\rceil\right)$$

✐ Exercise: how many page I/O operations does it take to sort an 8 GB file now?

Assume a page size of 8 kB. Available buffer space is $B = 1000$.

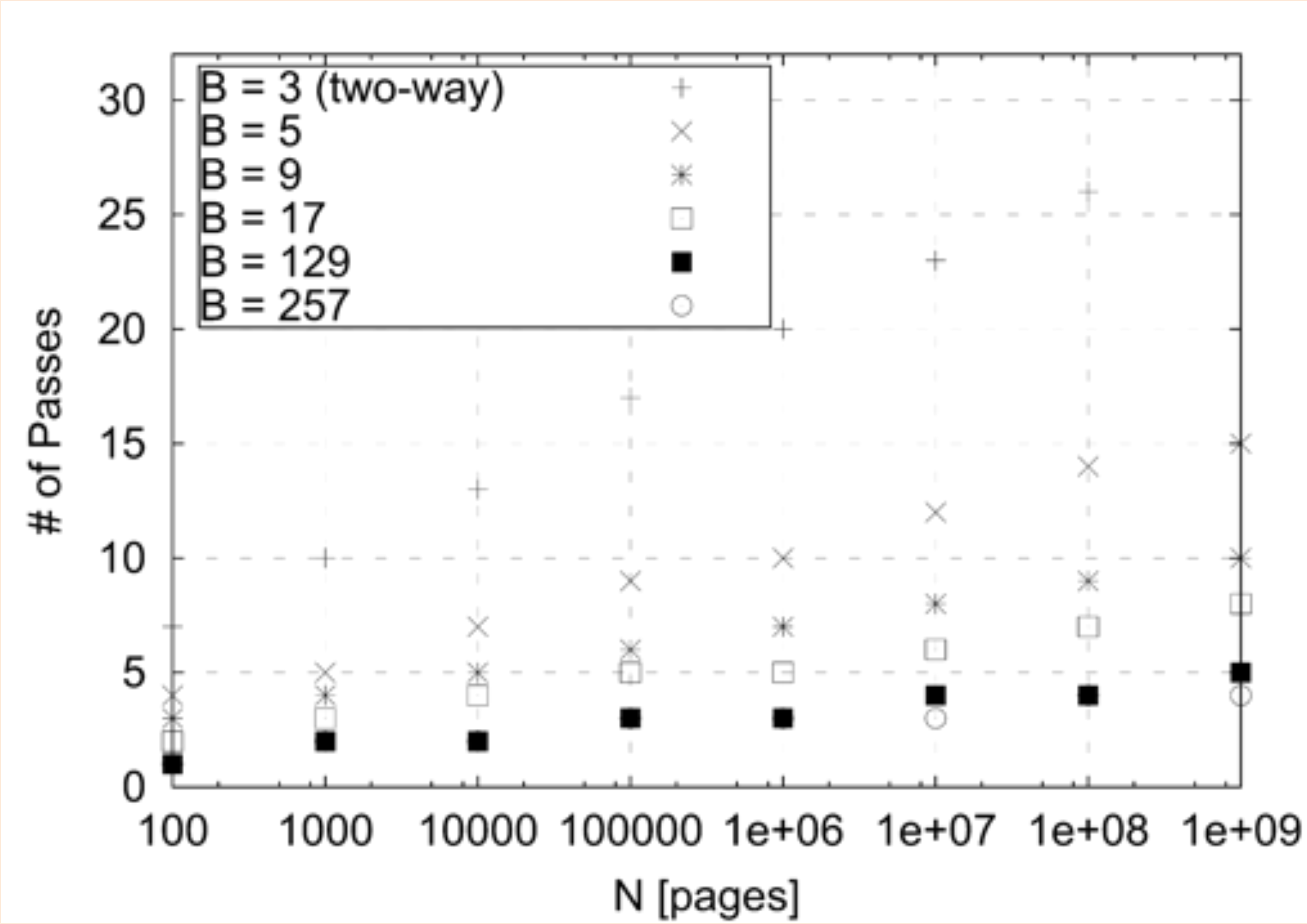
External Merge Sort

N	$B = 3$	$B = 5$	$B = 9$	$B = 17$	$B = 129$	$B = 257$
100	7	4	3	2	1	1
1000	10	5	4	3	2	2
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

Number of Passes of External Merge Sort

External Merge Sort

Number of passes for buffers of size B = 3, 5, ..., 257



External Merge Sort

✎ Exercise: I/O access pattern

Sorting N pages with B buffer pages requires

$$2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil {}^{N}/_{B} \rceil \rceil)$$

page I/O operations. What is the access pattern of these I/O operations?

Blocked I/O

- I/O pattern can be improved by using **blocked I/O**
  - allocate b pages for each input (instead of just one)
  - read **blocks of b pages** at once during the **merge** phases
- Trade-off between I/O cost and number of I/O operations
  - **reduces** I/O cost per page by a factor of ≈ b
  - reading blocks of pages **decreases the fan-in**, which increases the number of passes and therefore the number of I/O operations
- Total number of page I/O operations

$$2 \cdot N \cdot \left(1 + \left\lceil \log_{\lfloor B/b \rfloor - 1} \left\lceil {}^{N}/_{B} \right\rceil \right\rceil \right)$$

- In practice, main memory sizes are typically large enough to sort files with **just one merge pass**, even with blocked I/O

Blocked I/O

N	$B = 1000$	$B = 5000$	$B = 10{,}000$	$B = 50{,}000$
100	1	1	1	1
1000	1	1	1	1
10,000	2	2	1	1
100,000	3	2	2	2
1,000,000	3	2	2	2
10,000,000	4	3	3	2
100,000,000	5	3	3	2
1,000,000,000	5	4	3	3

Number of Passes of External Merge Sort with Block Size $b = 32$

Blocked I/O

✎ Exercise: How long does it take to sort 8 GB (counting I/O cost only)?

Assume 1000 buffer pages of 8 kB each, 8.5 ms average seek time, 60 MB/s transfer rate, and blocks of $b = 32$ pages.

Without blocked I/O

With blocked I/O

CPU Load

- External merge sort reduces the I/O load, but is considerably **more CPU intensive**
  - since I/O cost dominates the overall cost, this price is acceptable
- Consider the (B – 1)-**way merge** during passes 1, 2, …
  - to pick the next record to be moved to the output buffer, B – 2 comparisons need to be performed

Example: Comparisons for B – 1 = 4, θ = <

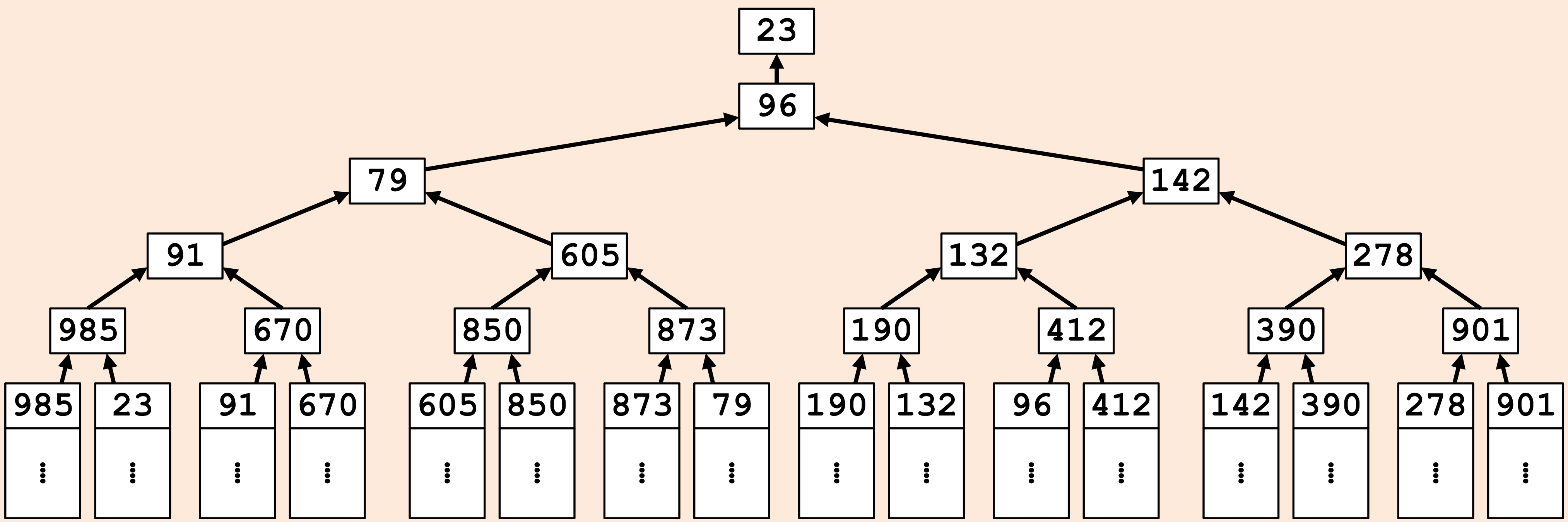
$$\left\{\begin{array}{l} 087\ 503\ 504\ \ldots \\ 170\ 908\ 994\ \ldots \\ 154\ 426\ 653\ \ldots \\ 612\ 613\ 700\ \ldots \end{array}\right. \rightarrow 087 \qquad \left\{\begin{array}{l} 503\ 504\ \ldots \\ 170\ 908\ 994\ \ldots \\ 154\ 426\ 653\ \ldots \\ 612\ 613\ 700\ \ldots \end{array}\right. \rightarrow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

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

CPU Load

- Number of comparisons can be reduced using a **selection tree**
  - "tree of losers" (D. Knuth: "The Art of Computer Programming", vol. 3)
  - this optimization cuts the number of comparisons down to $\log_2(B-1)$
  - for buffer sizes B >> 100 this is a considerable improvement

Example: Selection tree, read bottom-up



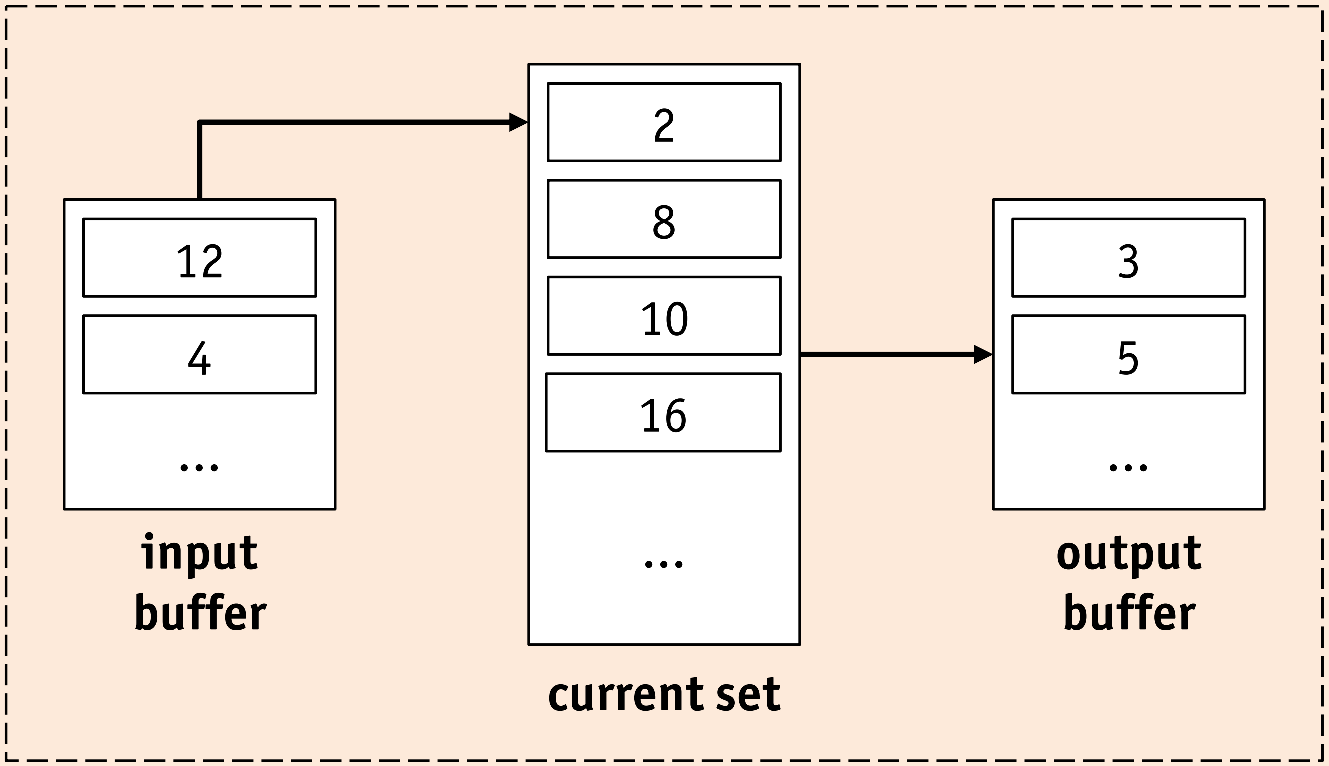
Minimizing the Number of Runs

- Number of initial runs **determines** number of required passes
  - initial runs are the files "run␣0␣r" written in Pass 0
  - because of $2 \cdot N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$, $r = 0, \ldots, \lceil N/B \rceil - 1$
- Reducing the number of initial runs is a **desirable optimization**
- **Replacement sort** is an example of such an optimization
  - **cut down** the number of $\lceil N/B \rceil$ initial runs in Pass 0
  - produce initial runs with **more than** B **pages**

Minimizing the Number of Runs

Replacement sort

Assume a buffer pool with *B* pages. Two pages are dedicated **input** and **output buffers.** The remaining *B* – 2 pages are called the **current set.**



Minimizing the Number of Runs

Replacement sort

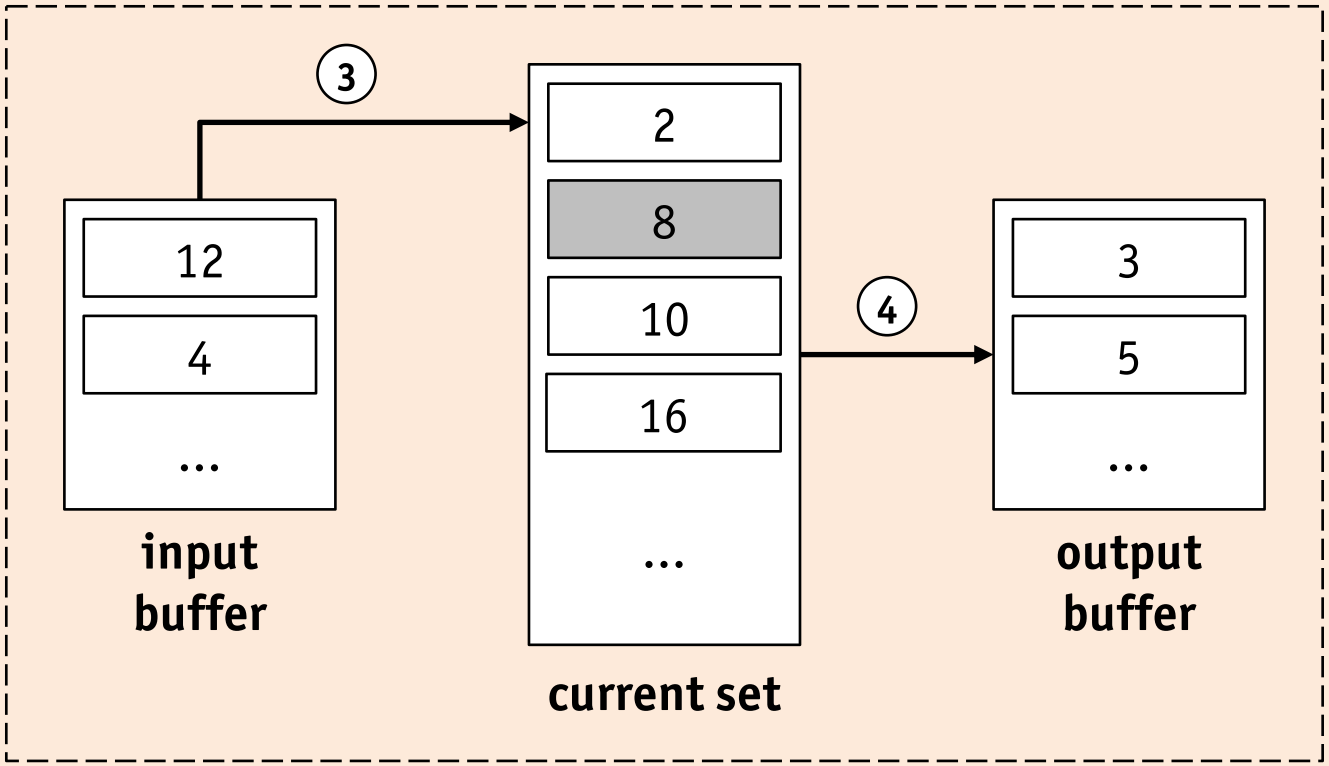
1. Open an empty run file for writing.
2. Load next page of file to be sorted into input buffer. If input file is exhausted, go to 4.
3. While there is space in the current set, move a record from input buffer to current set (if the input buffer is empty, reload it at 2.)
4. In current set, pick record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

- Remark
  - of course, Step 4 of replacement sort will benefit from techniques like the **selection tree**, especially if B – 2 (size of current set) is large

Minimizing the Number of Runs

Example

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



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

Minimizing the Number of Runs

✎ Exercise: tracing replacement sort

Assume B = 6, i.e., a current set size of 4. The input files contains records with **`INTEGER`** key values

503 087 512 061 908 170 897 275 426 154 509 612

Write a trace of replacement sort by filling out the table below, mark the end of the current run by ⟨**`EOR`**⟩. The current set has already been populated at Step 3.

current set				output
503	087	512	061	

Minimizing the Number of Runs

Length of initial runs

The trace of replacement sort suggests that the length of the initial runs indeed increases. In the example, the length of the first run is $8 \approx 2 \cdot B$, i.e., twice the size of the current set.

Exercise: analyzing the length of initial runs

How would you determine the average length of initial runs created by replacement sort in general?

CPU Load

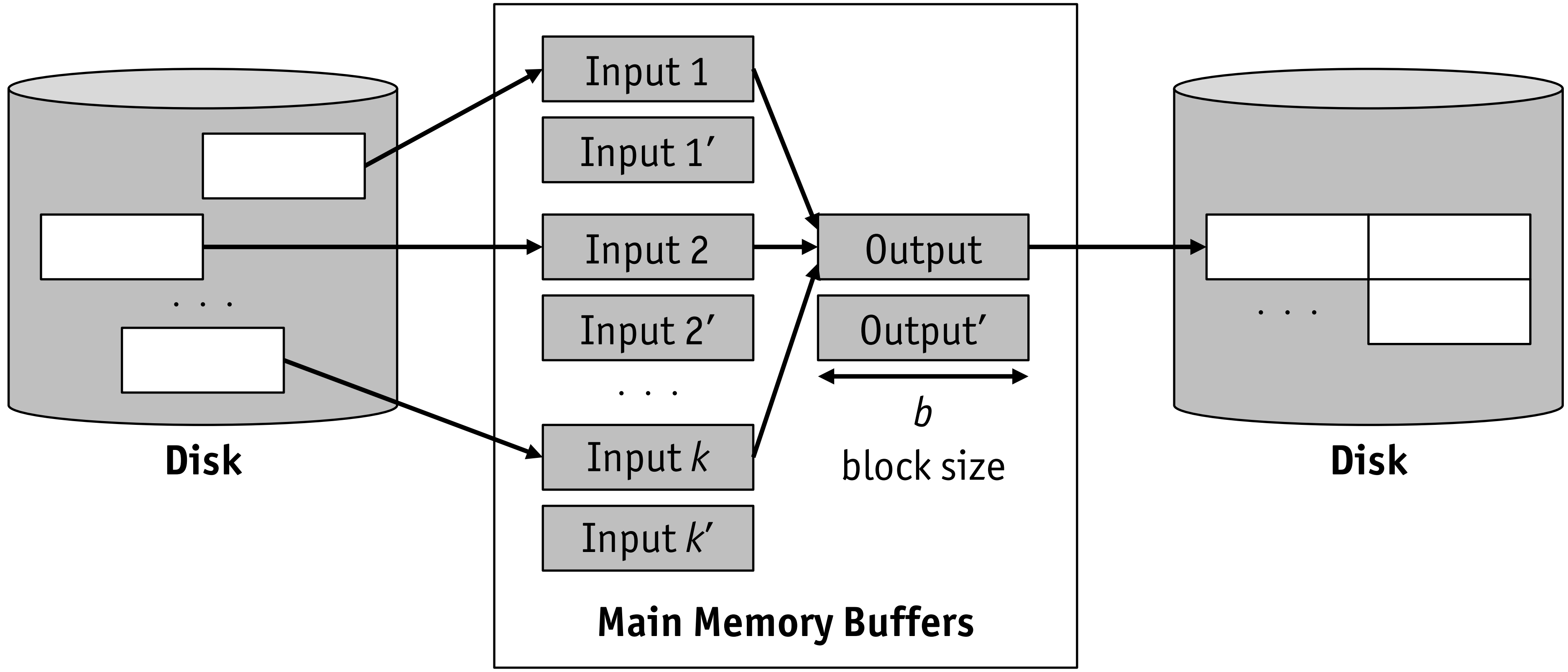
- External merge sort follows a **divide and conquer** principle
  - results in a number of **independent (sub-)tasks**
  - **execute tasks in parallel** in a parallel and distributed DBMS or exploit multi-core parallelism on modern CPUs
- To minimize **query response time**, CPU should never remain idle
  - avoid wait for input buffer to be **reloaded**
  - avoid wait for output buffer to be **appended** to current run

Double buffering

To avoid CPU waits, **double buffering** can be used.

1. create a second **shadow buffers** for each input and output buffer
2. CPU **switches** to the “double” buffer, once original buffers is empty/full
3. original buffer is reloaded by **asynchronously** initiating an I/O operation
4. CPU **switches** back to original buffer, once “double” buffer is empty/full, etc.

CPU Load



✎ Exercise: latency vs. throughput

Double buffering minimizes query response time. Does it also impact query throughput?

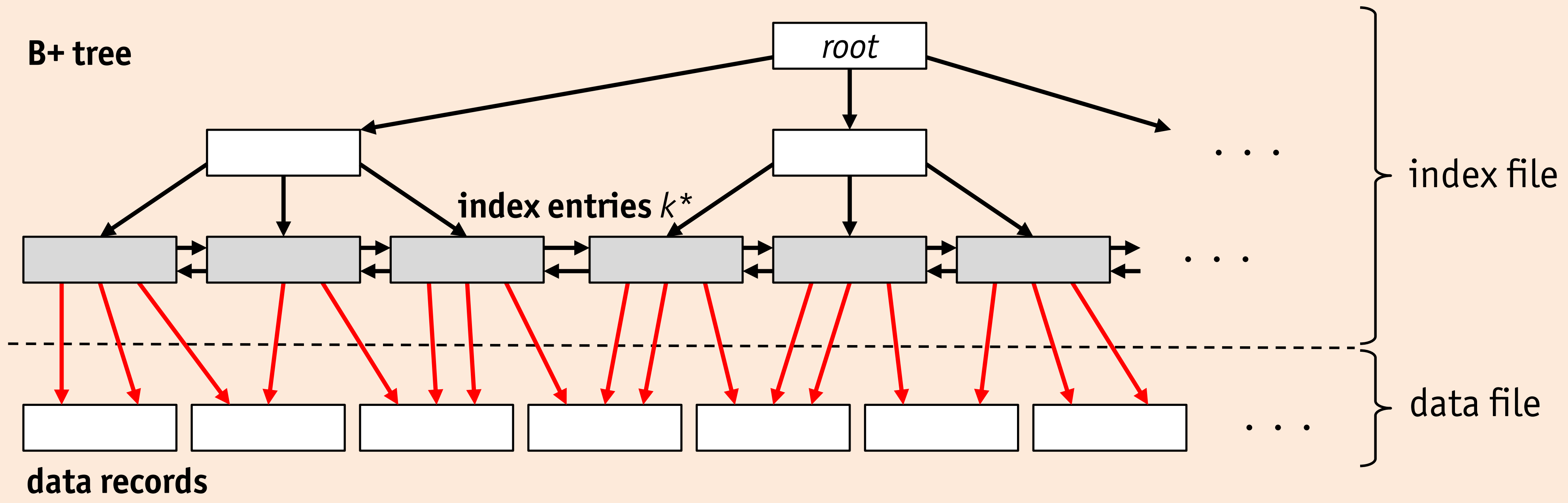
Using B+ Trees for Sorting

- Suppose that a B+ tree index matches a sorting task
  - B+ tree organized over key *k* with ordering θ
  - using the index and abandoning external sorting **may** be better
- Decision whether to use the index or not depends on the nature of the index
  - if index is clustered or unclustered
  - index entries used (variants ❶, ❷ and ❸)

Using B+ Trees for Sorting

- B+ tree index is **clustered**
  - data file itself is already θ-sorted
  - suffices to read all N pages of the file, regardless of the variant of index entries used

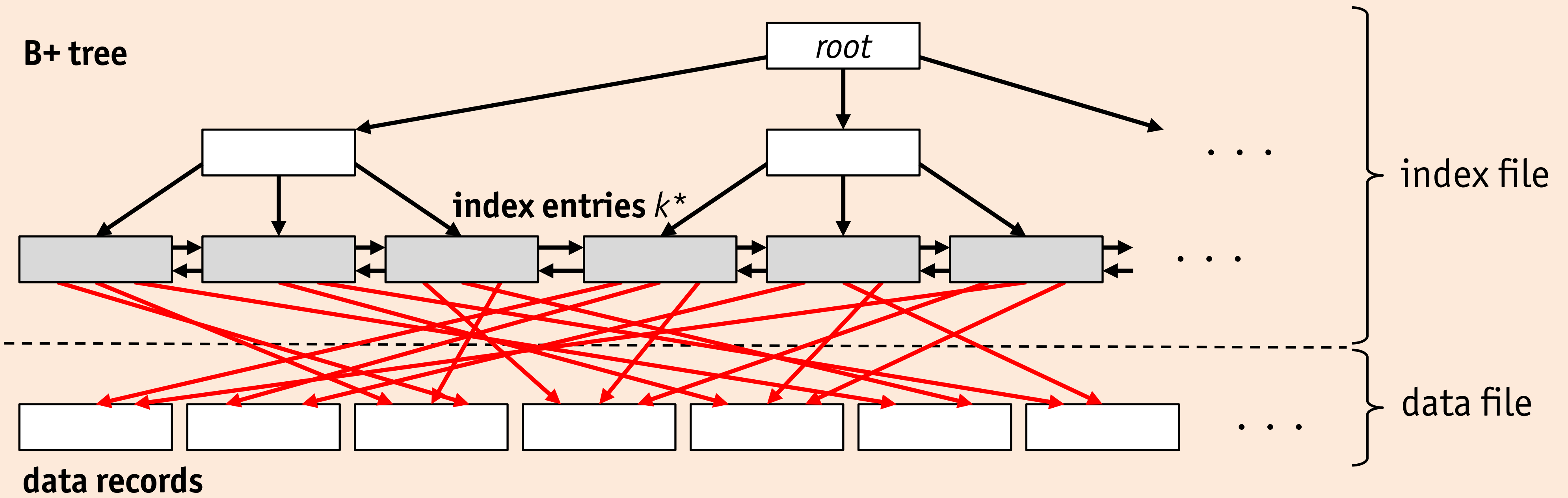
Clustered B+ tree index



Using B+ Trees for Sorting

- B+ tree index is **unclustered**
  - in the worst case, one page I/O operation has to be initiated per record (not per page) in the file!
  - therefore, do not use an unclustered index for sorting

Unclustered B+ tree index



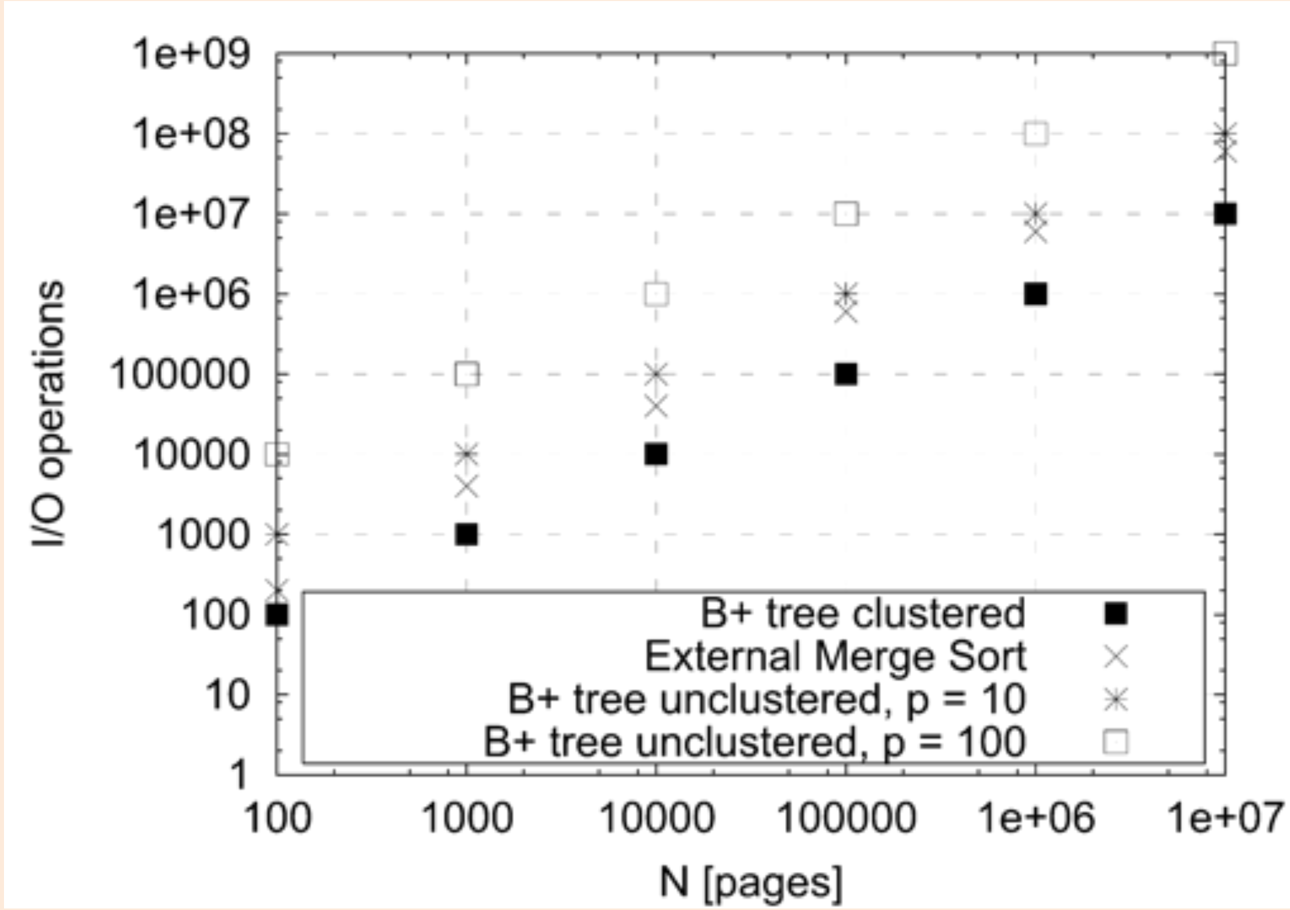
Using B+ Trees for Sorting

Expected page I/O operations for sorting

Let p denote the number of data records per page (typical values are p = 10, …, 1000). The expected number of page I/O operations to sort using an unclustered B+ tree index is therefore $\boldsymbol{p \cdot N}$ (worst case)

Assumptions in plot

- available buffer space for sorting is $B = 257$ pages
- ignore I/O to traverse B+ tree as well as its sequence set



- Even for modest file sizes, sorting by using an unclustered B+ tree index is **clearly inferior** to external sorting.

Sorting in Commercial DBMS

The Real World

- **IBM DB2**, **Microsoft SQL Server**, and **Oracle**
  - all systems use external merge sort
  - all systems use asynchronous I/O and prefetching
  - none of these systems uses optimization that produces runs larger than available memory, in part because it is difficult to implement it efficiently in the presence of variable-length records
- **IBM DB2**
  - uses a separate area of memory to do sorting
  - uses radix sort as in-memory sorting algorithm
- **Oracle**
  - uses a separate area of memory to do sorting
  - uses insertion sort as in-memory sorting algorithm
- **Microsoft SQL Server**
  - uses buffer pool frames for sorting
  - uses merge sort as in-memory sorting algorithm

Universität
Konstanz

Database System Architecture and Implementation

TO BE CONTINUED...