

- Two-Way Merge Sort
- External Merge Sort
- Comparisons
- Replacement Sort
- B<sup>+</sup>-trees for Sorting

# Chapter 7

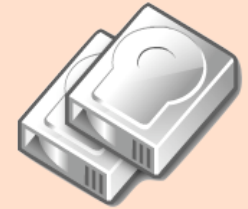
## External Sorting

Sorting Tables Larger Than Main Memory

*Architecture and Implementation of Database Systems*

Winter 2010/11





Two-Way Merge Sort  
External Merge Sort  
Comparisons  
Replacement Sort  
B<sup>+</sup>-trees for Sorting

## Challenges lurking behind a SQL query

aggregation

```
SELECT C.CUST_ID, C.NAME, SUM(O.TOTAL) AS REVENUE
FROM CUSTOMERS AS C, ORDERS AS O
WHERE C.ZIPCODE BETWEEN 8000 AND 8999
AND C.CUST_ID = O.CUST_ID
GROUP BY C.CUST_ID
ORDER BY C.CUST_ID, C.NAME
```

grouping

selection

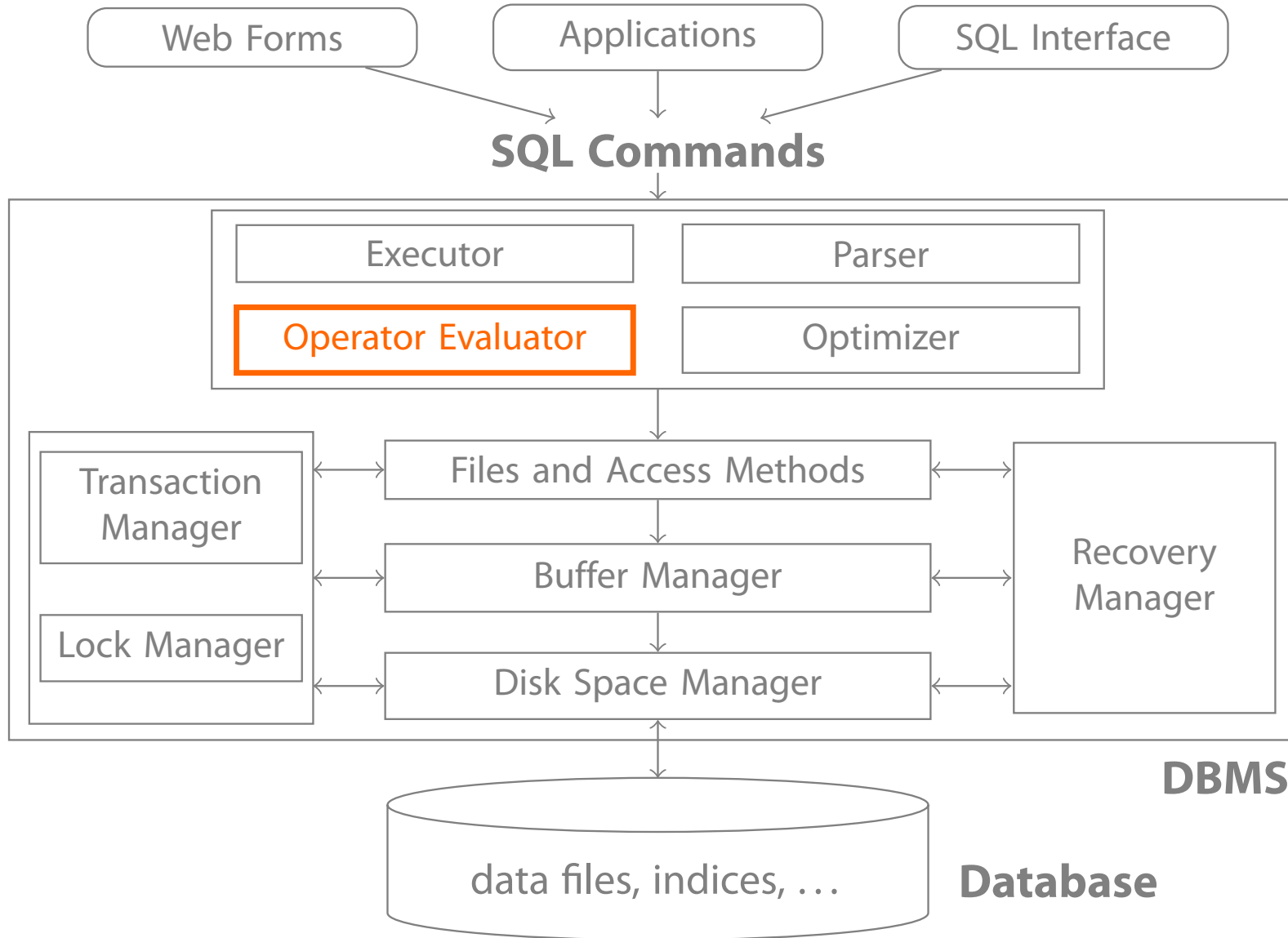
join

sorting

A DBMS **query processor** needs to perform a number of tasks

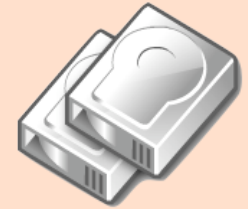
- with **limited memory resources**,
- over **large amounts of data**,
- yet **as fast as possible**.

# Query Processing



External Sorting

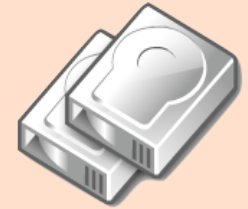
Torsten Grust



Query Processing

Sorting

- Two-Way Merge Sort
- External Merge Sort
- Comparisons
- Replacement Sort
- $B^+$ -trees for Sorting



## Query plans and operators

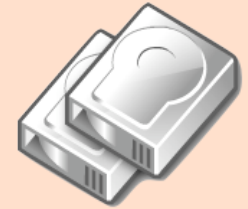
### Query Processing

#### Sorting

Two-Way Merge Sort  
External Merge Sort  
Comparisons  
Replacement Sort  
 $B^+$ -trees for Sorting

- DBMS does *not* execute a query as a large monolithic block but rather provides a number of specialized routines, the **query operators**.
  - Operators are “plugged together” to form a network of operators, a **plan**, that is capable of evaluating a given query.
  - Each operator is carefully implemented to perform a specific task well (*i.e.*, time- and space-efficient).
- 
- **Now:** Zoom in on the details of the implementation of one of the most basic and important operators: **sort**.

# Query Processing: Sorting



- **Sorting** stands out as a useful operation, explicit or implicit:

## Explicit sorting via the SQL ORDER BY clause

```
1      SELECT  A,B,C
2      FROM    R
3      ORDER BY A
```

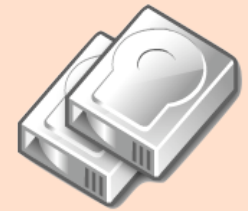
## Implicit sorting, e.g., for duplicate elimination

```
1      SELECT DISTINCT A,B,C
2      FROM          R
```

## Implicit sorting, e.g., to prepare equi-join

```
1      SELECT R.A,S.Y
2      FROM  R,S
3      WHERE R.B = S.X
```

- Further:  
Grouping via GROUP BY, B<sup>+</sup>-tree bulk loading, sorted *rid* scans after access to unclustered indexes, ...



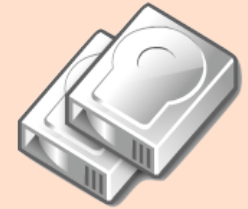
## Sorting

- A file is **sorted** with respect to **sort key**  $k$  and **ordering**  $\theta$ , if for any two records  $r_{1,2}$  with  $r_1$  preceding  $r_2$  in the file, we have that their corresponding keys are in  $\theta$ -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:  $k = (A, B)$ ,  $\theta = <$ :

$$r_1 < r_2 \quad \Leftrightarrow \quad r_1.A < r_2.A \vee \\ (r_1.A = r_2.A \wedge r_1.B < r_2.B) \ .$$



- As it is a principal goal not to restrict the file sizes a DBMS can handle, we face a fundamental problem:

*How can we sort a file of records whose **size exceeds the available main memory space** (let alone the available buffer manager space) by far?*

- Approach the task in a two-phase fashion:
  - ① Sorting a file of arbitrary size is possible even if **three pages** of buffer space is all that is available.
  - ② Refine this algorithm to make effective use of larger and thus more realistic buffer sizes.
- As we go along, consider a number of further optimizations in order to **reduce the overall number of required page I/O operations**.

# Two-Way Merge Sort

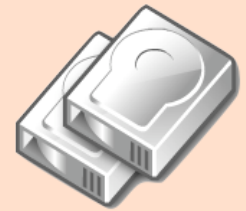
We start with **two-way merge sort**, which can sort files of arbitrary size with only **three pages** of buffer space.

## Two-way merge sort

Two-way merge sort sorts a file with  $N = 2^k$  pages in multiple **passes**, each of them producing a certain number of sorted sub-files 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  $k$**  leaves only one run left, the sorted overall result.

During each pass, we consult every page in the file. Hence,  $k \cdot N$  page reads and  $k \cdot N$  page writes are required to sort the file.





# Basic Two-Way Merge Sort Idea

**Pass 0** (Input:  $N = 2^k$  unsorted pages; Output:  $2^k$  sorted runs)

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 runs; 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.

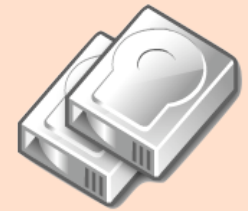
⋮

**Pass  $n$**  (Input:  $2^{k-n+1}$  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).

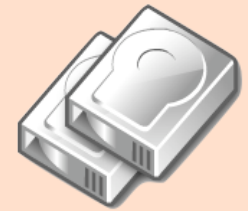
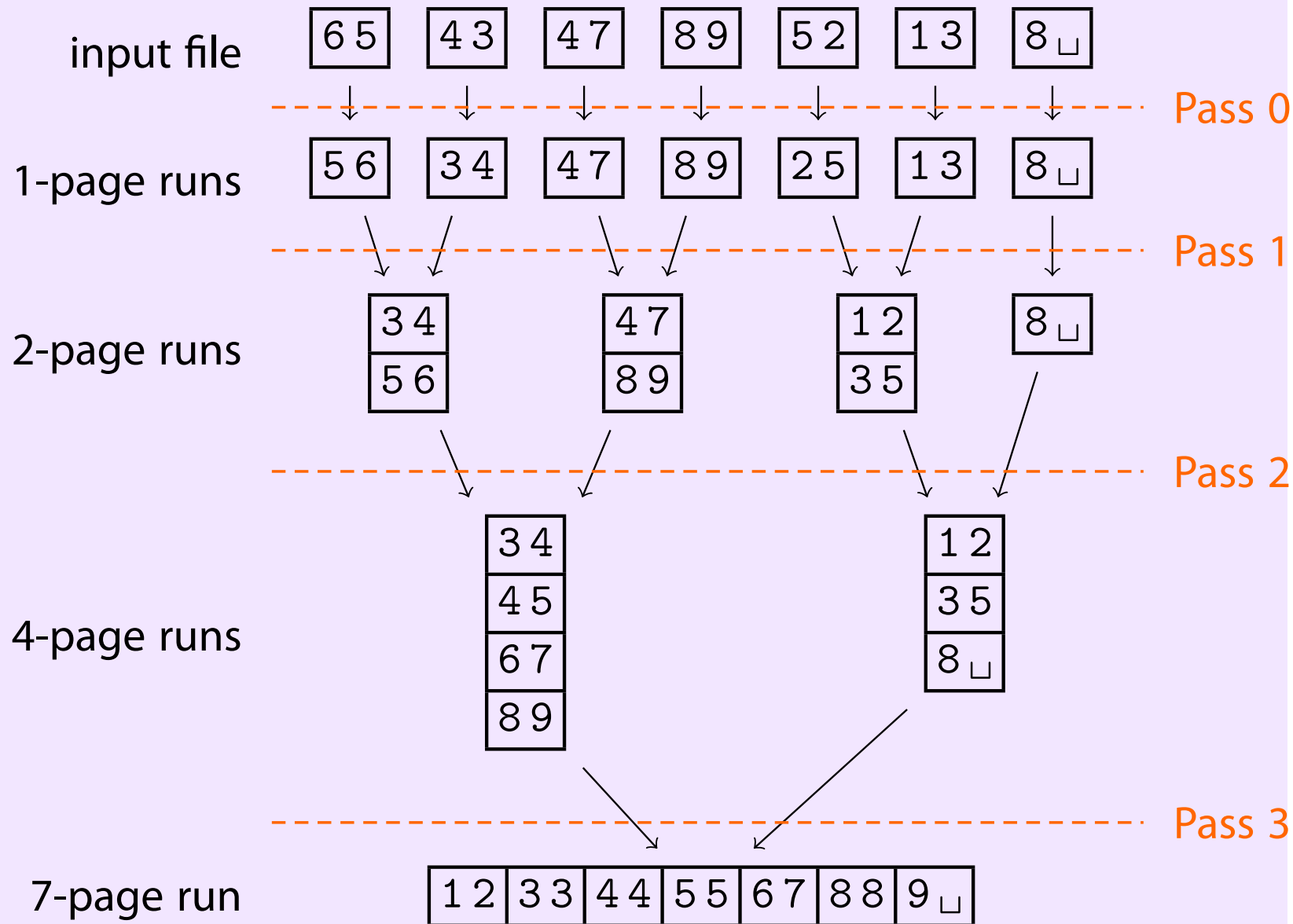
This pass requires **three pages** of buffer space.

⋮

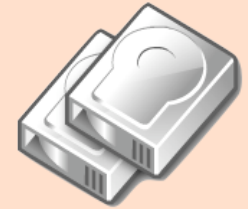


# Two-way Merge Sort: Example

Example (7-page file, two records per page, keys  $k$  shown,  $\theta = <$ )



# Two-Way Merge Sort: Algorithm

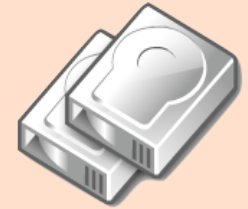


## Two-way merge sort, $N = 2^k$

```
1 Function: two_way_merge_sort (file, N)
  /* Pass 0: create N sorted single-page runs
    (in-memory sort) */
2 foreach page p in file do
3   | read p into memory, sort it, write it out into a new run;
  /* next k passes merge pairs of runs, until only one
    run is left */
4 for n in 1...k do
5   | for r in 0... $2^{k-n} - 1$  do
6     | merge runs  $2 \cdot r$  and  $2 \cdot r + 1$  from previous pass into a
7     | new run, reading the input runs one page at a time;
8     | delete input runs  $2 \cdot r$  and  $2 \cdot r + 1$ ;
  result  $\leftarrow$  last output run;
```

Each merge requires **three buffer frames** (two to read the two input files and one to construct output pages).

## Two-Way Merge Sort: I/O Behavior



- To sort a file of  $N$  pages, in each pass we read  $N$  pages, sort/merge, and write  $N$  pages out again:

$$2 \cdot N \quad \text{I/O operations per pass}$$

- Number of passes:

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

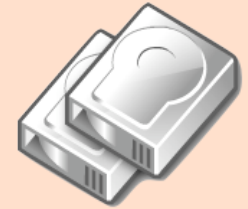
- **Total number of I/O operations:**

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

 **How many I/Os does it take to sort an 8 GB file?**

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

# External Merge Sort

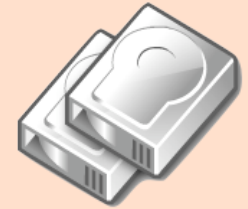


- So far we have “voluntarily” used only three pages of buffer space.

*How could we **make effective use of a significantly larger buffer page pool** (of, say,  $B$  frames)?*

- Basically, there are two knobs we can turn and tune:
  - 1 **Reduce the number of initial runs** by using the full buffer space during the in-memory sort.

# External Merge Sort



- So far we have “voluntarily” used only three pages of buffer space.

*How could we **make effective use of a significantly larger buffer page pool** (of, say,  $B$  frames)?*

- Basically, there are two knobs we can turn and tune:
  - 1 **Reduce the number of initial runs** by using the full buffer space during the in-memory sort.
  - 2 **Reduce the number of passes** by merging more than two runs at a time.

## Reducing the Number of Initial Runs

With  $B$  frames available in the buffer pool, **we can read  $B$  pages at a time during Pass 0** and sort them in memory (↗ slide 9):

**Pass 0** (Input:  $N$  unsorted pages; Output:  $\lceil N/B \rceil$  sorted runs)

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

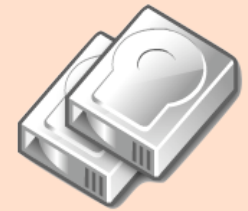
The **number of initial runs** determines the **number of passes** we need to make (↗ slide 12):

⇒ **Total number of I/O operations:**

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

 **How many I/Os does it take to sort an 8 GB file now?**

Again, assume 8 KB pages. Available buffer space is  $B = 1,000$ .



## Reducing the Number of Passes

With  $B$  frames available in the buffer pool, we can **merge  $B - 1$  pages at a time** (leaving one frame as a write buffer).

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

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

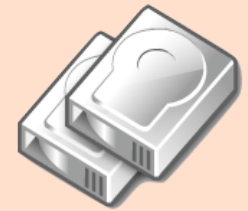
With  $B$  pages of buffer space, we can do a  $(B - 1)$ -**way merge**.

⇒ **Total number of I/O operations:**

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

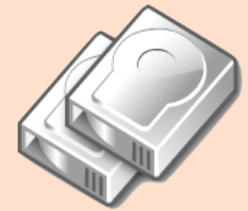
 **How many I/Os does it take to sort an 8 GB file now?**

Again, assume 8 KB pages. Available buffer space is  $B = 1,000$ .

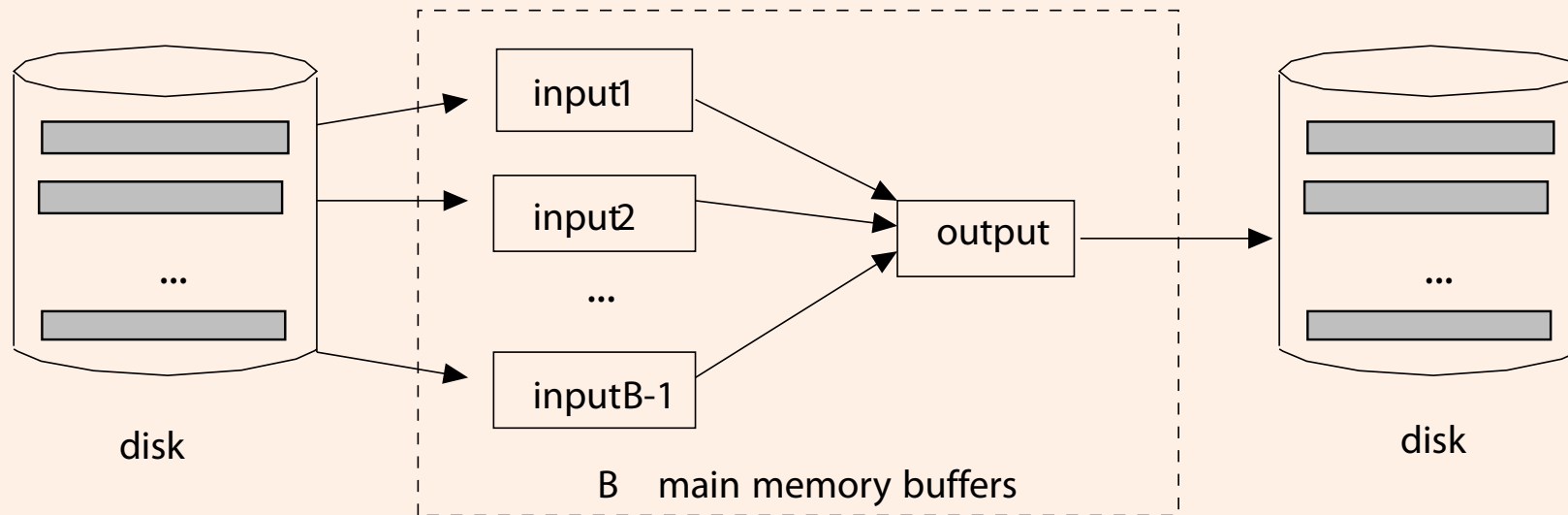




# Reducing the Number of Passes



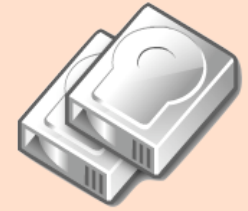
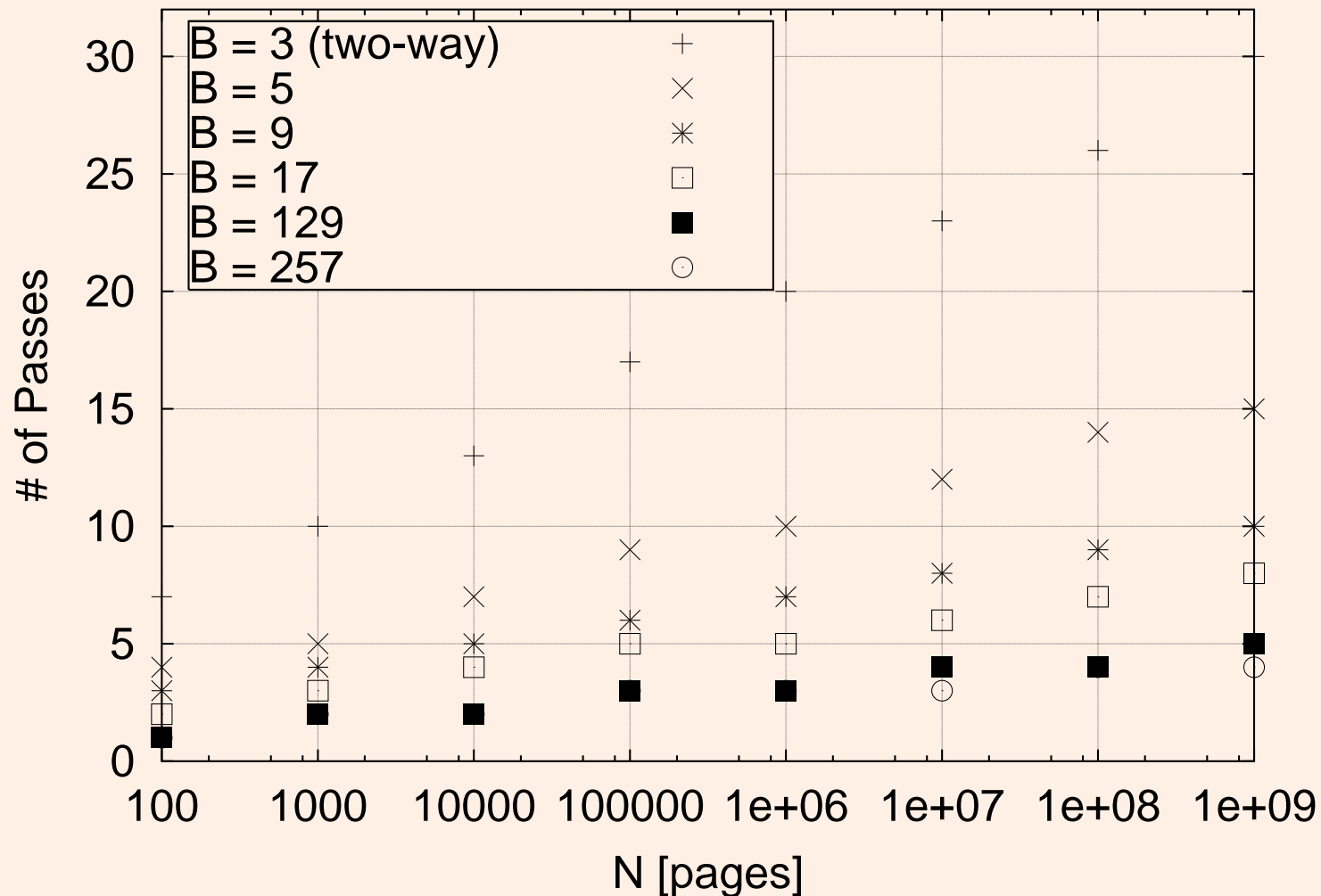
## $(B - 1)$ -way merge using a buffer of $B$ pages



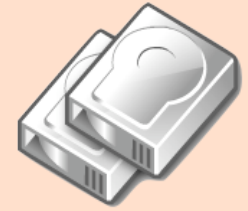
## External Sorting: I/O Behavior

- The I/O savings in comparison to two-way merge sort ( $B = 3$ ) can be substantial:

# of passes for buffers of size  $B = 3, 5, \dots, 257$



# External Sorting: I/O Behavior



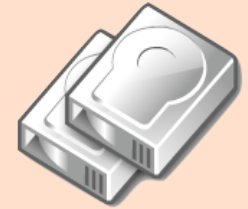
- Sorting  $N$  pages with  $B$  buffer frames requires

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

I/O operations.

 What is the access pattern of these I/Os?

# External Sorting: I/O Behavior



- Sorting  $N$  pages with  $B$  buffer frames requires

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

I/O operations.

## What is the access pattern of these I/Os?

- In Pass 0, we read chunks of size  $B$  **sequentially**.
- Everything else is **random access** due to the  $B - 1$  way merge.  
(Which of the  $B - 1$  runs will contribute the next record...?)

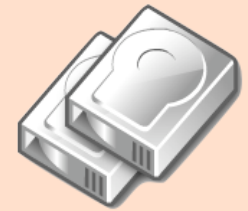
## Blocked I/O

We could improve the I/O pattern by reading **blocks** of, say,  $b$  pages at once during the **merge** phases.

- Allocate  $b$  pages for each input (instead of just one).
- **Reduces per-page I/O cost** by a factor of  $\approx b$ .
- The price we pay is a **decreased fan-in** (resulting in an increased number of passes and more I/O operations).
- In practice, main memory sizes are typically large enough to sort files with **just one merge pass**, even with blocked I/O.

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

Assume 1,000 buffer pages of 8 KB each, 8.5 ms average seek time.



## Blocked I/O

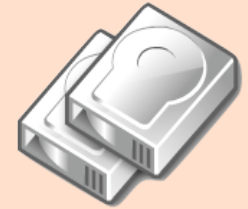
We could improve the I/O pattern by reading **blocks** of, say,  $b$  pages at once during the **merge** phases.

- Allocate  $b$  pages for each input (instead of just one).
- **Reduces per-page I/O cost** by a factor of  $\approx b$ .
- The price we pay is a **decreased fan-in** (resulting in an increased number of passes and more I/O operations).
- In practice, main memory sizes are typically large enough to sort files with **just one merge pass**, even with blocked I/O.

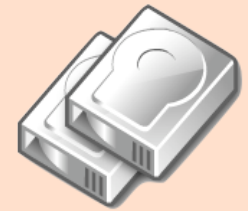
### How long does it take to sort 8 GB (counting only I/O cost)?

Assume 1,000 buffer pages of 8 KB each, 8.5 ms average seek time.

- Without blocked I/O:  $\approx 4 \cdot 10^6$  disk seeks (10.5 h) + transfer of  $\approx 6 \cdot 10^6$  disk pages (13.6 min)
- With blocked I/O ( $b = 32$  page blocks):  $\approx 6 \cdot 32,800$  disk seeks (28.1 min) + transfer of  $\approx 8 \cdot 10^6$  disk pages (18.1 min)



# External Merge Sort: CPU Load and Comparisons

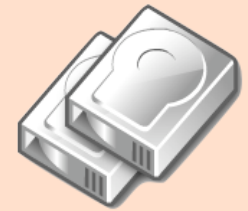


- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  $(B - 1)$ -**way merge** during passes  $1, 2, \dots$ :  
To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

## Example (Comparisons for $B - 1 = 4, \theta = <$ )

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

# External Merge Sort: CPU Load and Comparisons



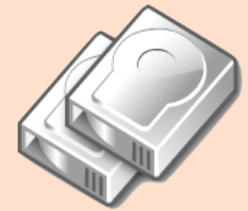
- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  $(B - 1)$ -**way merge** during passes 1, 2, ...:  
To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

## Example (Comparisons for $B - 1 = 4, \theta = <$ )

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



# External Merge Sort: CPU Load and Comparisons

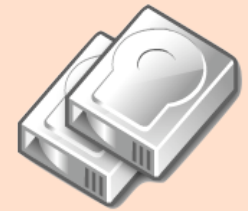


- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  $(B - 1)$ -**way merge** during passes 1, 2, ...:  
To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

## Example (Comparisons for $B - 1 = 4, \theta = <$ )

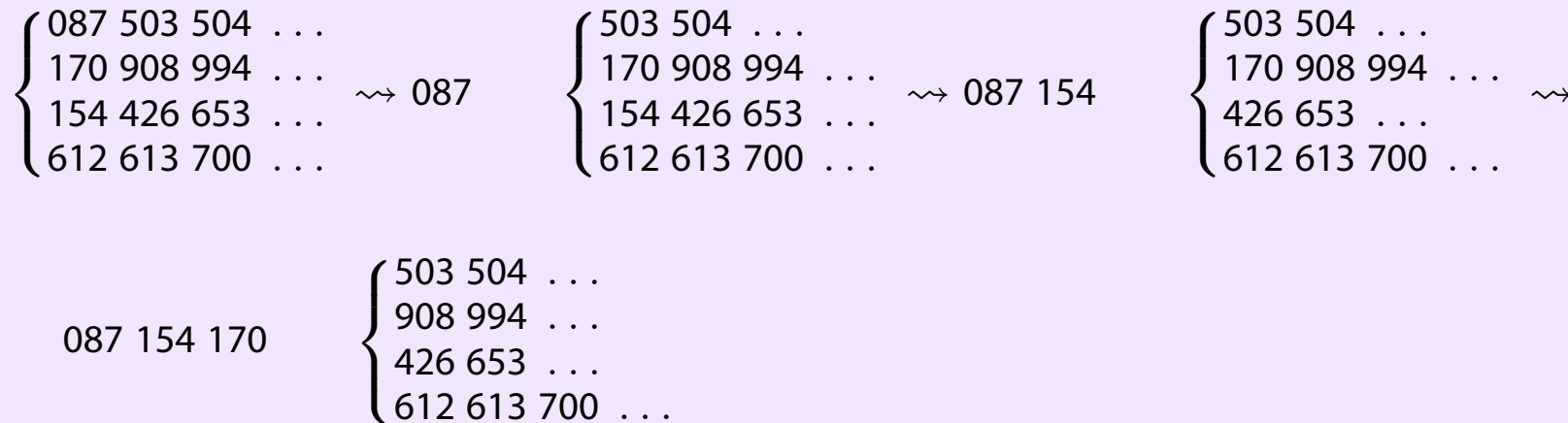
$$\left\{ \begin{array}{l} 087 \ 503 \ 504 \ \dots \\ 170 \ 908 \ 994 \ \dots \\ 154 \ 426 \ 653 \ \dots \\ 612 \ 613 \ 700 \ \dots \end{array} \right. \rightsquigarrow 087 \quad \left\{ \begin{array}{l} 503 \ 504 \ \dots \\ 170 \ 908 \ 994 \ \dots \\ 154 \ 426 \ 653 \ \dots \\ 612 \ 613 \ 700 \ \dots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

# External Merge Sort: CPU Load and Comparisons

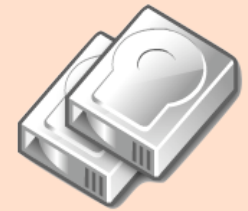


- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  $(B - 1)$ -**way merge** during passes 1, 2, ...:  
To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

## Example (Comparisons for $B - 1 = 4, \theta = <$ )

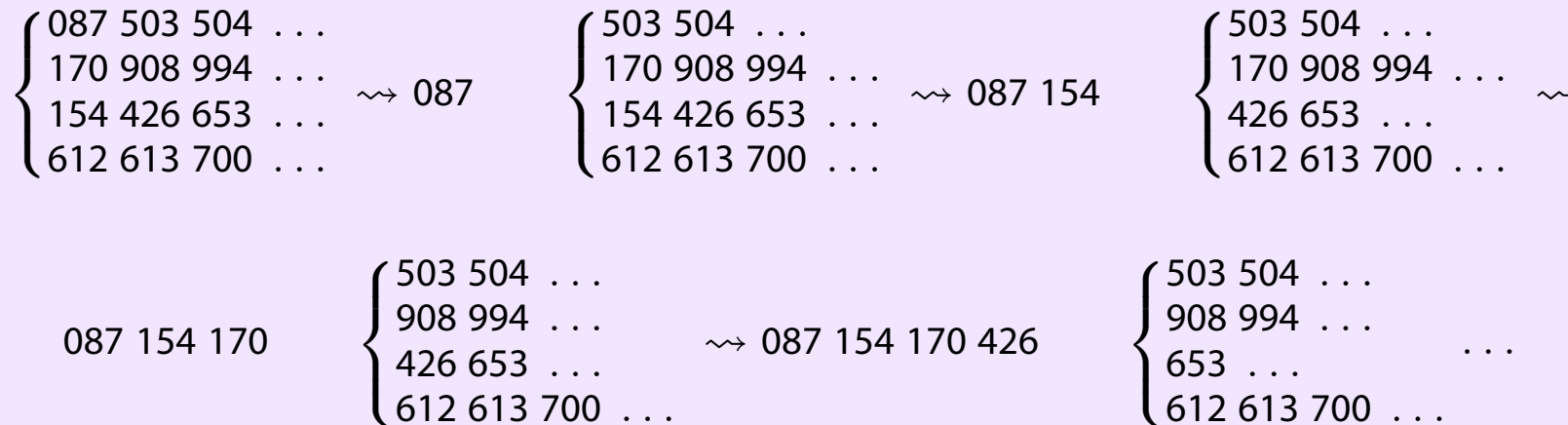


# External Merge Sort: CPU Load and Comparisons

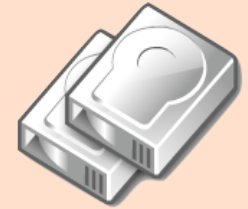


- External merge sort reduces the I/O load, but is considerably **CPU intensive**.
- Consider the  $(B - 1)$ -**way merge** during passes 1, 2, ...:  
To pick the next record to be moved to the output buffer, we need to perform  $B - 2$  comparisons.

## Example (Comparisons for $B - 1 = 4, \theta = <$ )



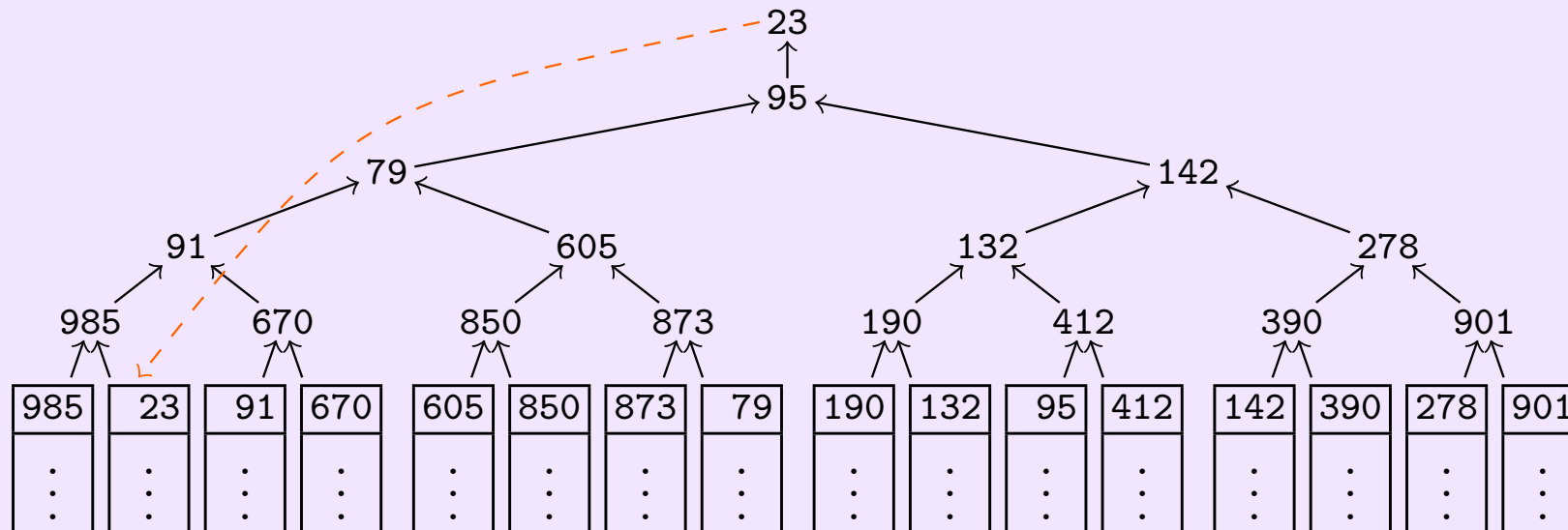
# Selection Trees



Choosing the next record from  $B - 1$  (or  $B/b - 1$ ) input runs can be quite CPU intensive ( $B - 2$  comparisons).

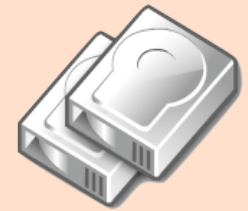
- Use a **selection tree** to reduce this cost.
- E.g., “tree of losers” (↗ D. Knuth, TAOCP, vol. 3):

## Example (Selection tree, read bottom-up)



- This cuts the number of comparisons down to  $\log_2 (B - 1)$ .

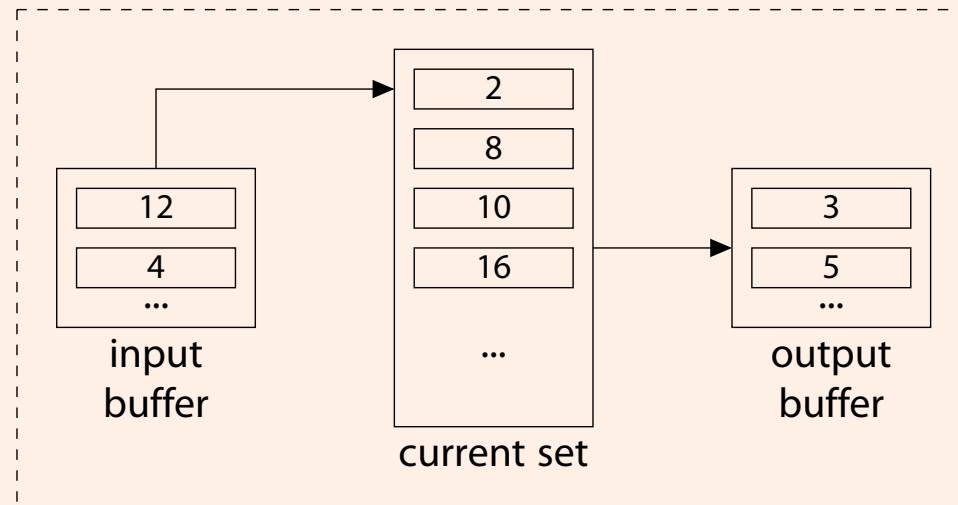
# Further Reducing the Number of Initial Runs

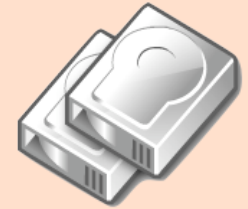


- **Replacement sort** can help to further cut down the number of initial runs  $\lceil N/B \rceil$ : try to **produce initial runs with more than  $B$  pages**.

## Replacement sort

- 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

### Query Processing

#### Sorting

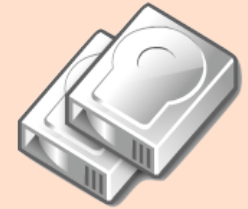
Two-Way Merge Sort  
External Merge Sort  
Comparisons  
Replacement Sort  
B<sup>+</sup>-trees for Sorting

- 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.<sup>1</sup> 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.

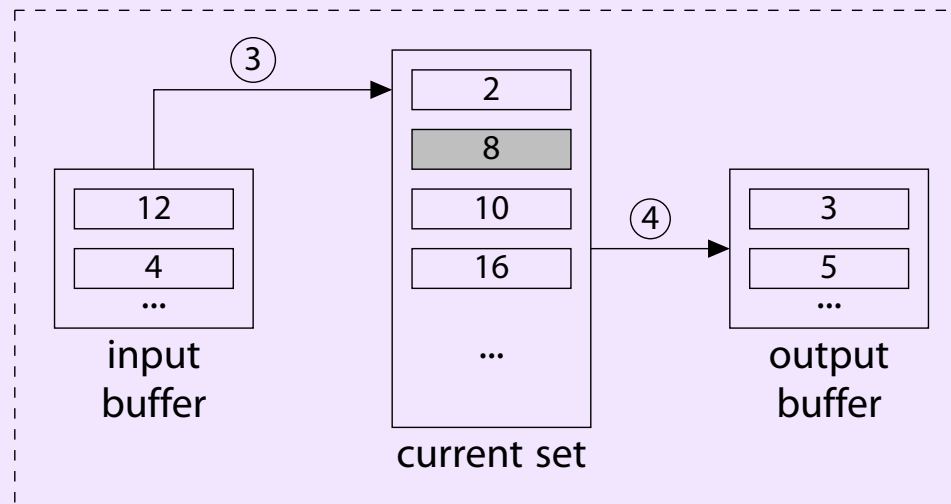
---

<sup>1</sup>If output buffer is empty, define  $k_{out} = -\infty$ .

# Replacement Sort

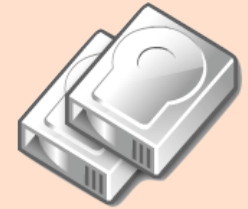


Example (Record with key  $k = 8$  will be the next to be moved into the output buffer; current  $k_{out} = 5$ )



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

# Replacement Sort



## Tracing replacement sort

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

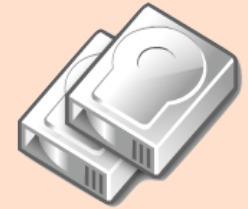
503 087 512 061 908 170 897 275 426 154 509 612 .

Write a trace of replacement sort by filling out the table below, mark the end of the current run by  $\langle \text{EOR} \rangle$  (the current set has already been populated at step 3):

current set				output
503	087	512	061	



# Replacement Sort

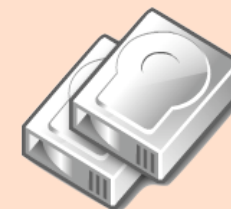


- Step 4 of replacement sort will benefit from techniques like selection tree, esp. if  $B - 2$  is large.
- The replacement sort trace suggests that the length of the initial runs indeed increases. In the example: first run length  $7 \approx$  **twice the size of the current set**.

## Length of initial runs?

Implement replacement sort to empirically determine initial run length or check the proper analysis ([↗ D. Knuth, TAOCP, vol. 3, p. 254](#)).

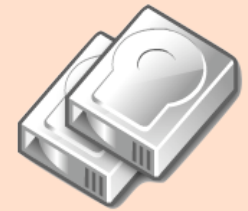
# External Sort: Remarks



- External sorting follows a **divide and conquer** principle.
  - This results in a number of **indepdendent (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**:

Create **shadow buffers** for the input and output buffers. Let the CPU switch to the “double” input buffer as soon as the input buffer is empty and **asynchronously initiate an I/O operation** to reload the original input buffer. Treat the output buffer similarly.

## (Not) Using B<sup>+</sup>-trees for Sorting



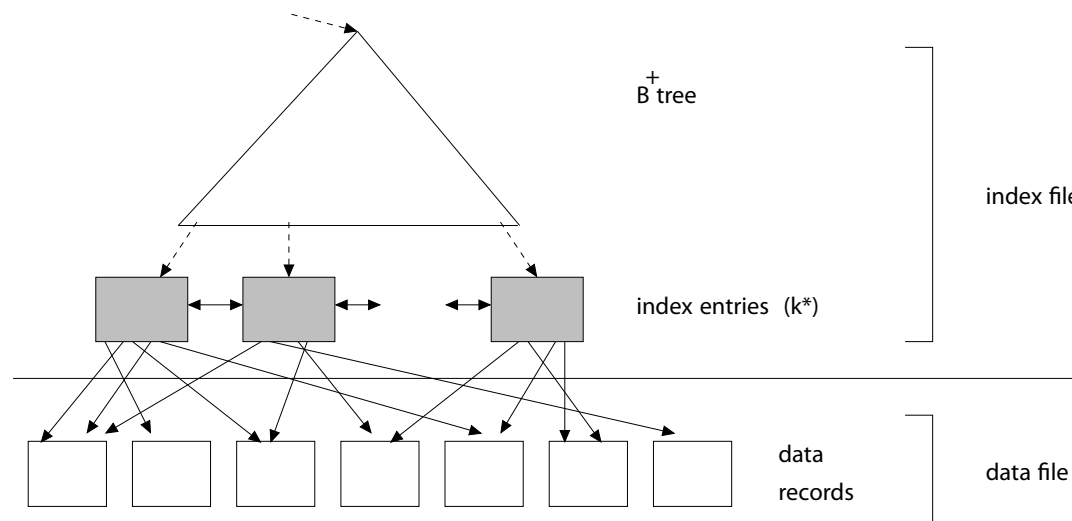
- If a B<sup>+</sup>-tree matches a sorting task (*i.e.*, B<sup>+</sup>-tree organized over key  $k$  with ordering  $\theta$ ), we *may* be better off to **access the index and abandon external sorting**.

① If the B<sup>+</sup>-tree is **clustered**, then

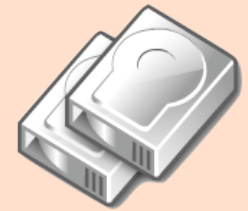
- the data file itself is already  $\theta$ -sorted,
- ⇒ simply sequentially read the sequence set (or the pages of the data file).

② If the B<sup>+</sup>-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.



# (Not) Using B<sup>+</sup>-tree for Sorting



## Query Processing

### Sorting

- Two-Way Merge Sort
- External Merge Sort
- Comparisons
- Replacement Sort
- B<sup>+</sup>-trees for Sorting**

- Let  $p$  denote the number of records per page (typically,  $p = 10, \dots, 1000$ ). Expected of I/O operations to sort via an *unclustered* B<sup>+</sup>-tree will thus be  $p \cdot N$ :

## Expected sort I/O operations (assume $B = 257$ )

