# Database Management Systems

Lecture 7

Evaluating Relational Operators

Query Optimization (II)

- running example - schema
  - Students (<u>SID: integer</u>, SName: string, Age: integer)
  - Courses (<u>CID: integer</u>, CName: string, Description: string)
  - Exams (<u>SID: integer, CID: integer</u>, EDate: date, Grade: integer, FacultyMember: string)

  - Students
    - every record has 50 bytes
    - there are 80 records / page
    - 500 pages of Students tuples
  - Courses
    - every record has 50 bytes
    - there are 80 records / page
    - 100 pages of Courses tuples

- running example - schema
  - Students (<u>SID: integer</u>, SName: string, Age: integer)
  - Courses (<u>CID: integer</u>, CName: string, Description: string)
  - Exams (<u>SID: integer, CID: integer</u>, EDate: date, Grade: integer, FacultyMember: string)

  - Exams
    - every record has 40 bytes
    - there are 100 records / page
    - 1000 pages of Exams tuples

* sorting

- can be explicitly required:
    - SELECT ... ORDER BY list
    - SELECT DISTINCT …
- used by operators like:
    - join
    - union
    - intersection
    - set-difference
    - grouping

* <u>sorting</u>

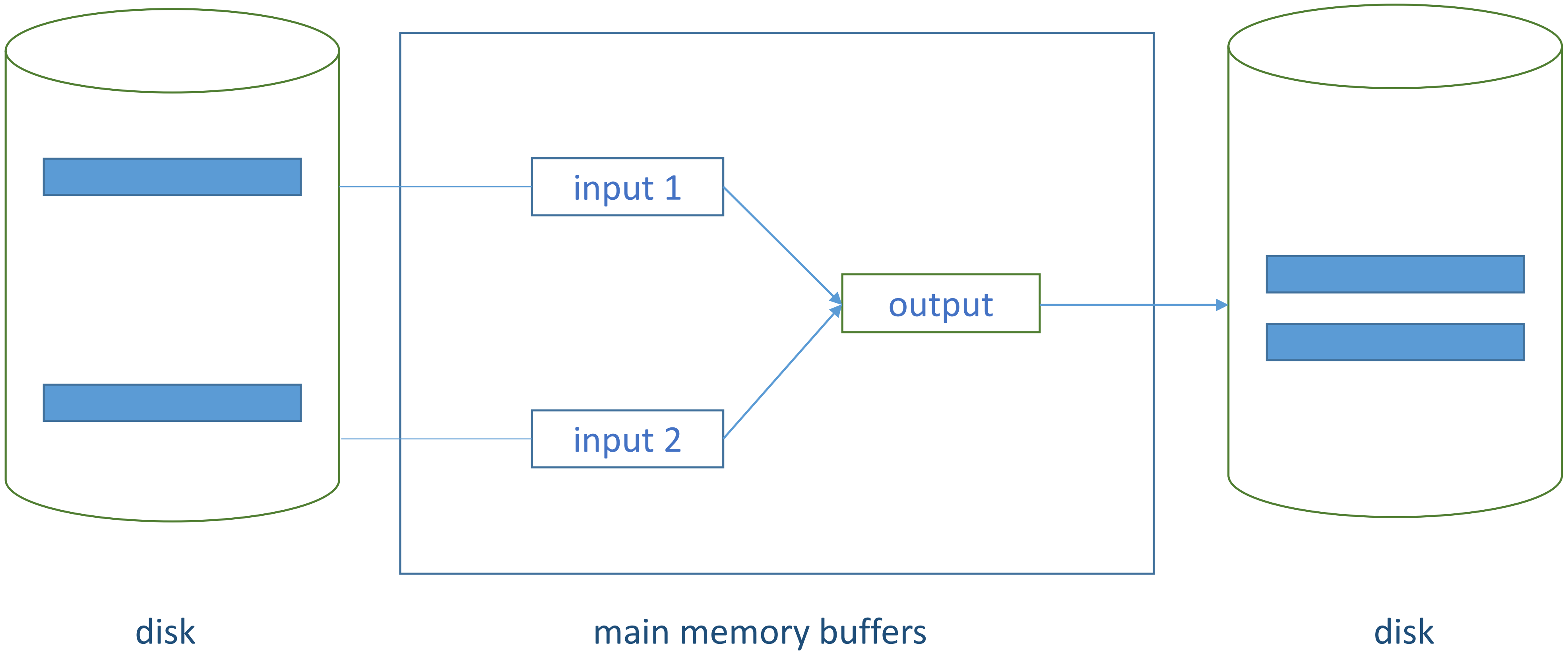
e.g., the user wants to sort the collection of Courses records by name

- if the data to be sorted fits into available main memory:
  - use an <u>internal sorting</u> algorithm (Quick Sort or any other in-memory sorting algorithm can be used to sort a collection of records that fits into main memory)

- if the data to be sorted doesn't fit into available main memory:
  - use an <u>external sorting</u> algorithm
  - such an algorithm:
    - minimizes the cost of accessing the disk
    - breaks the data collection into subcollections of records
    - sorts the subcollections; a sorted subcollection of records is called a *run*
    - writes runs to disk (into temporary files)
    - merges runs

# Simple Two-Way Merge Sort

- uses 3 buffer pages
- passes over the data multiple times
- can sort large data collections using a small amount of main memory



## Simple Two-Way Merge Sort

- pass 0:
  - for each page P in the data collection:
    - read in page P -> sort page P -> save page P to disk

  => 1-page runs (runs that are 1 page long)

  example: - read in the 1st page from Courses, sort the 80 records on it by course name, write out the sorted page to disk (i.e., a *run* that is one page long);

  - read in the 2nd page from Courses, sort the 80 records on it by course name, write out sorted page to disk;

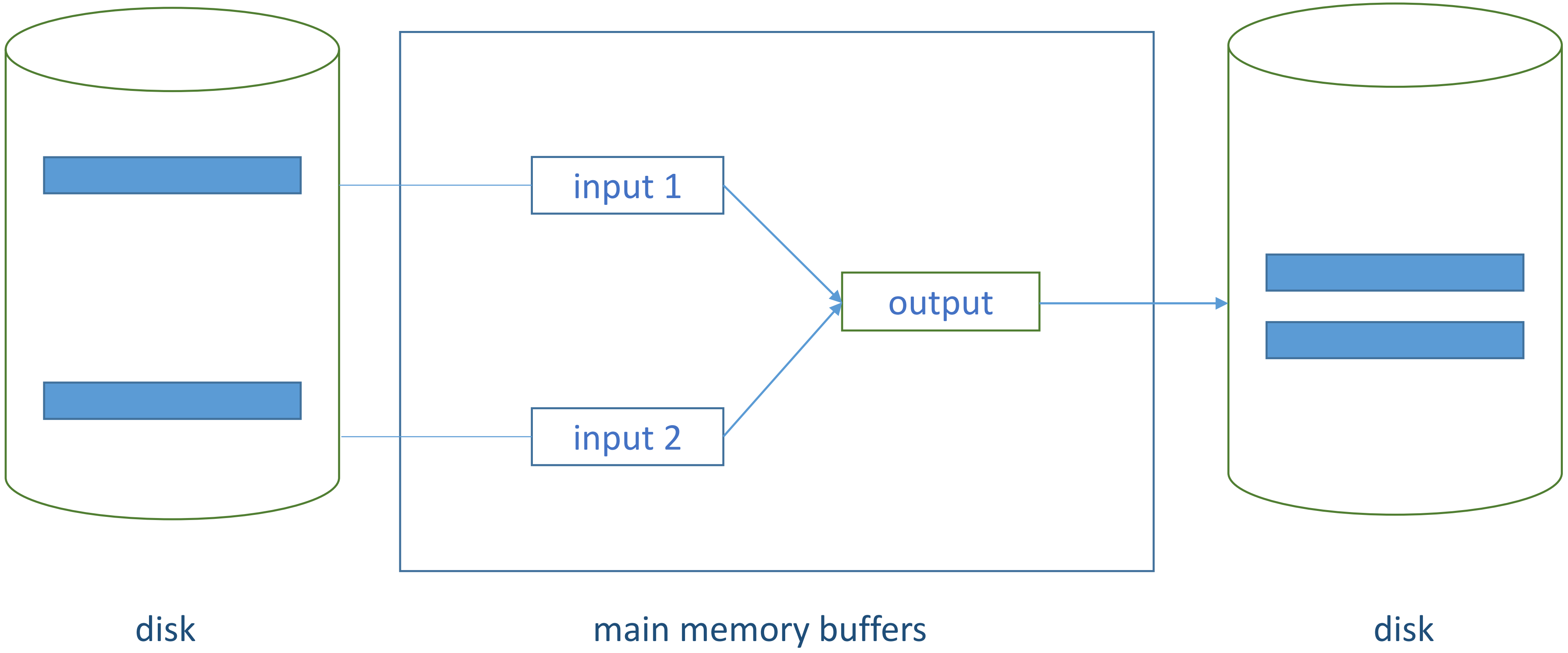
  …

  - read in the 100th page from Courses, sort the 80 records on it by course name, write out sorted page to disk

  => 100 1-page runs saved on disk

# Simple Two-Way Merge Sort

- passes 1, 2, … etc:
  - use 3 buffer pages
  - read and merge pairs of runs from the previous passes
  - produce runs that are twice as long



## Simple Two-Way Merge Sort

- e.g., pass 1 (remember, pass 0 produced 100 1-page runs):
  - read in 2 runs from pass 0 (i.e., two pages holding Courses records, each of them sorted in pass 0), using 2 buffer pages
    - merge these runs writing to the 3rd available buffer page (the *output* buffer); when the output buffer fills up, write it out to disk (i.e., write a page of 80 sorted records to disk)

    => a run that is 2 pages long (it contains 160 Courses records, sorted by name)
  - read in and merge the next 2 runs from pass 0 … => another run that is 2 pages long
  - continue while there are runs to be processed (read in and merged) from pass 0
  - at the end of pass 1 there are 50 2-page runs (each run consists of 2 pages holding 160 records sorted by course name)

# Simple Two-Way Merge Sort

- another example – sort data collection (file of records) with 7 pages:

| | | | | | | |
|---|---|---|---|---|---|---|
| 3, 4 | 6, 2 | 9, 4 | 8, 7 | 5, 6 | 3, 1 | 2 |
| page 1 | page 2 | ... | | | | |

- only the value of the key is displayed (the key on which the user wants to sort the collection, an integer number in the example)
- simplifying assumption that allows us to focus on the idea of the algorithm: a page can hold 2 records

- pass 0
  - read in the collection one page at a time
  - sort each page that is read in
  - write out each sorted page to disk

  => 7 sorted runs that are 1 page long:

| | | | | | | |
|---|---|---|---|---|---|---|
| 3, 4 | 2, 6 | 4, 9 | 7, 8 | 5, 6 | 1, 3 | 2 |

# Simple Two-Way Merge Sort

- runs at the end of pass 0:

| | | | | | | |
|---|---|---|---|---|---|---|
| 3, 4 | 2, 6 | 4, 9 | 7, 8 | 5, 6 | 1, 3 | 2 |

- pass 1
  - read in & merge pairs of runs from pass 0
  - produce runs that are twice as long
  - read in runs 3, 4 and 2, 6 :
    - merge the runs and write to the output buffer
    - write the output buffer to disk one page at a time

  => run

| | |
|---|---|
| 2, 3 | 4, 6 |

  - read in runs 4, 9 and 7, 8 :
    - merge the runs and write to the output buffer
    - write the output buffer to disk one page at a time

  => run

| | |
|---|---|
| 4, 7 | 8, 9 |

# Simple Two-Way Merge Sort

- runs at the end of pass 0: $\boxed{3, 4}$ $\boxed{2, 6}$ $\boxed{4, 9}$ $\boxed{7, 8}$ $\boxed{5, 6}$ $\boxed{1, 3}$ $\boxed{2}$

- pass 1
  - read in runs $\boxed{5, 6}$ and $\boxed{1, 3}$ …

    => run $\boxed{1, 3}\boxed{5, 6}$

  - read in run $\boxed{2}$ (the last run from pass 0) …

    => run $\boxed{2}$

  => 4 sorted runs that are 2 pages long (except for the last run):

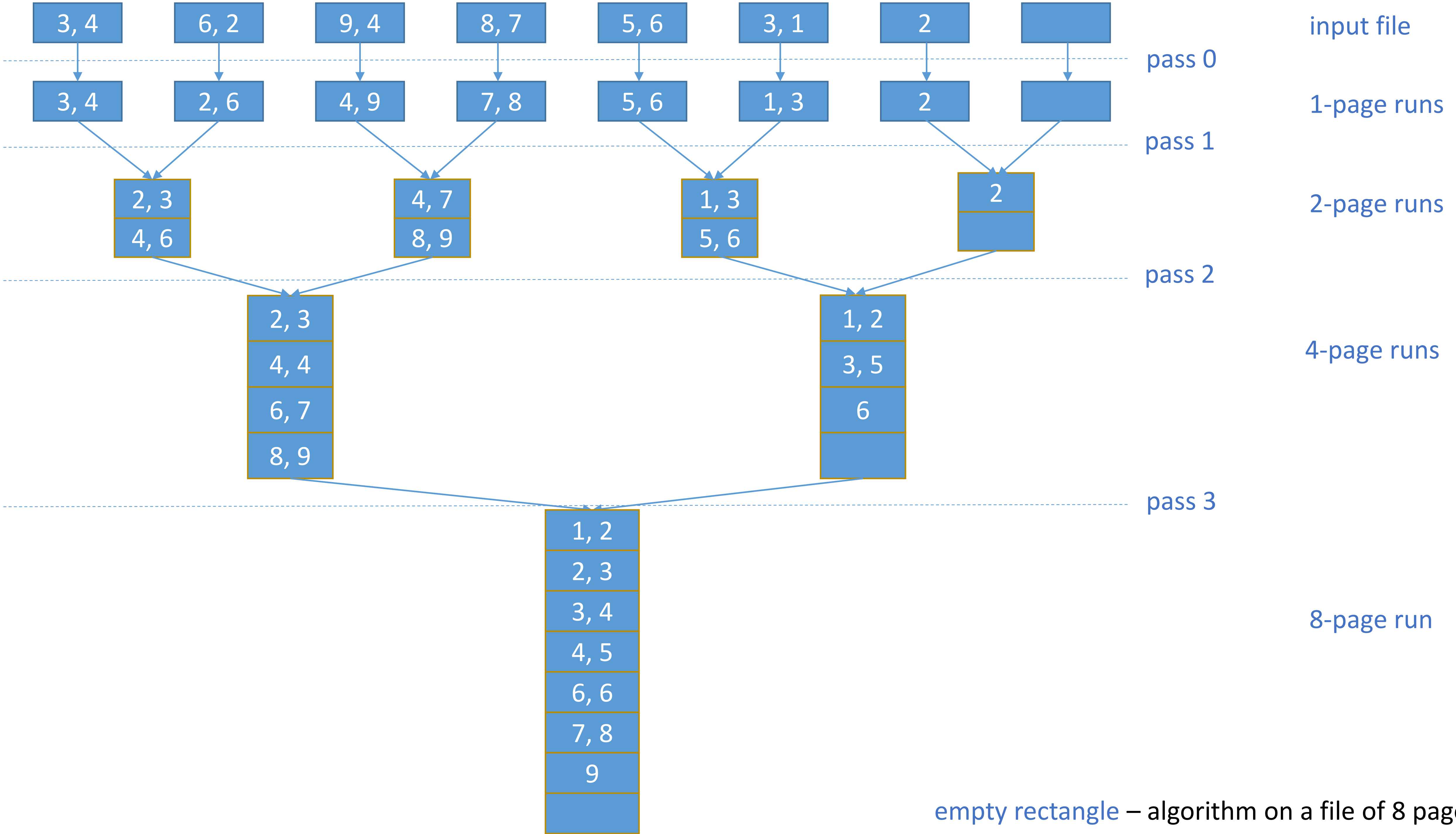
  $\boxed{2, 3}\boxed{4, 6}$ $\boxed{4, 7}\boxed{8, 9}$ $\boxed{1, 3}\boxed{5, 6}$ $\boxed{2}$

# Simple Two-Way Merge Sort

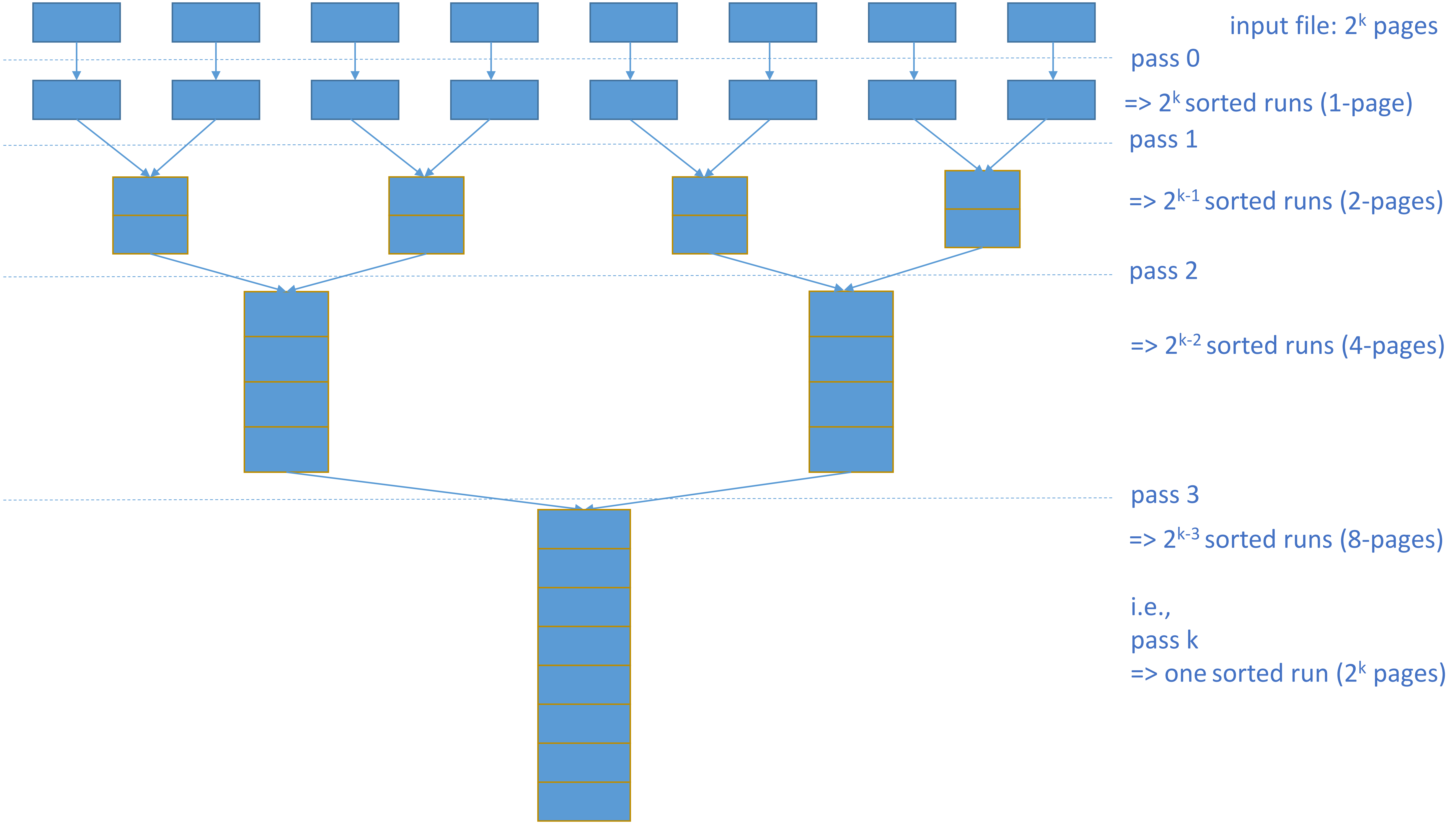
- pass 2
    - read in & merge pairs of runs from pass 1
    - produce runs that are twice as long

  …
- complete example, with all passes of the algorithm, on the next page ->

->

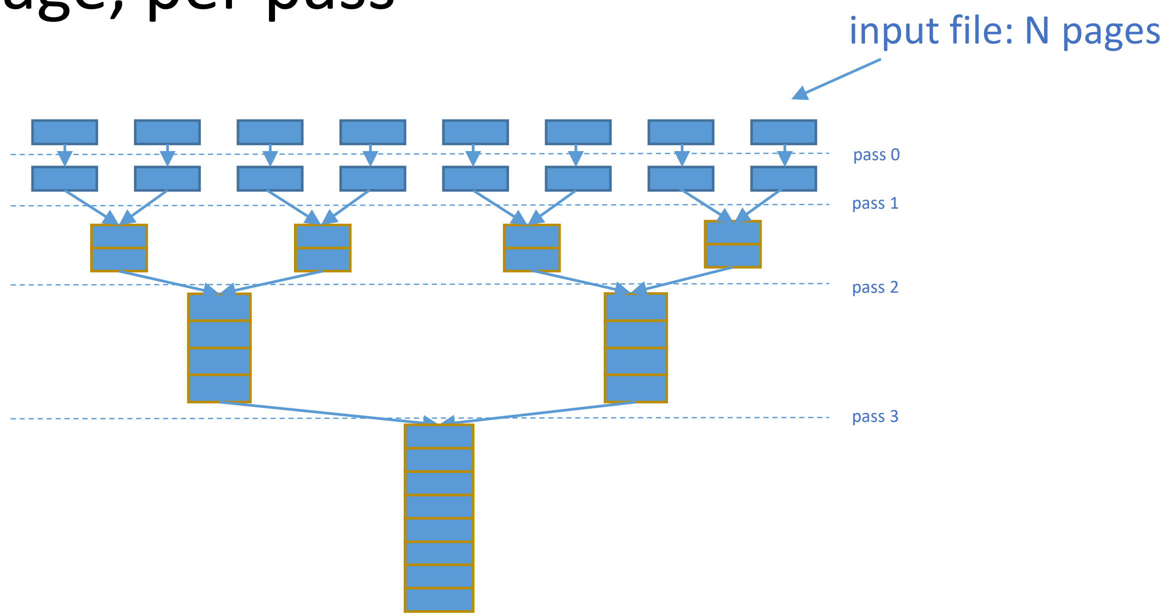



empty rectangle – algorithm on a file of 8 pages


input file: $2^k$ pages
pass 0
=> $2^k$ sorted runs (1-page)
pass 1
=> $2^{k-1}$ sorted runs (2-pages)
pass 2
=> $2^{k-2}$ sorted runs (4-pages)
pass 3
=> $2^{k-3}$ sorted runs (8-pages)
i.e.,
pass k
=> one sorted run ($2^k$ pages)

# Simple Two-Way Merge Sort

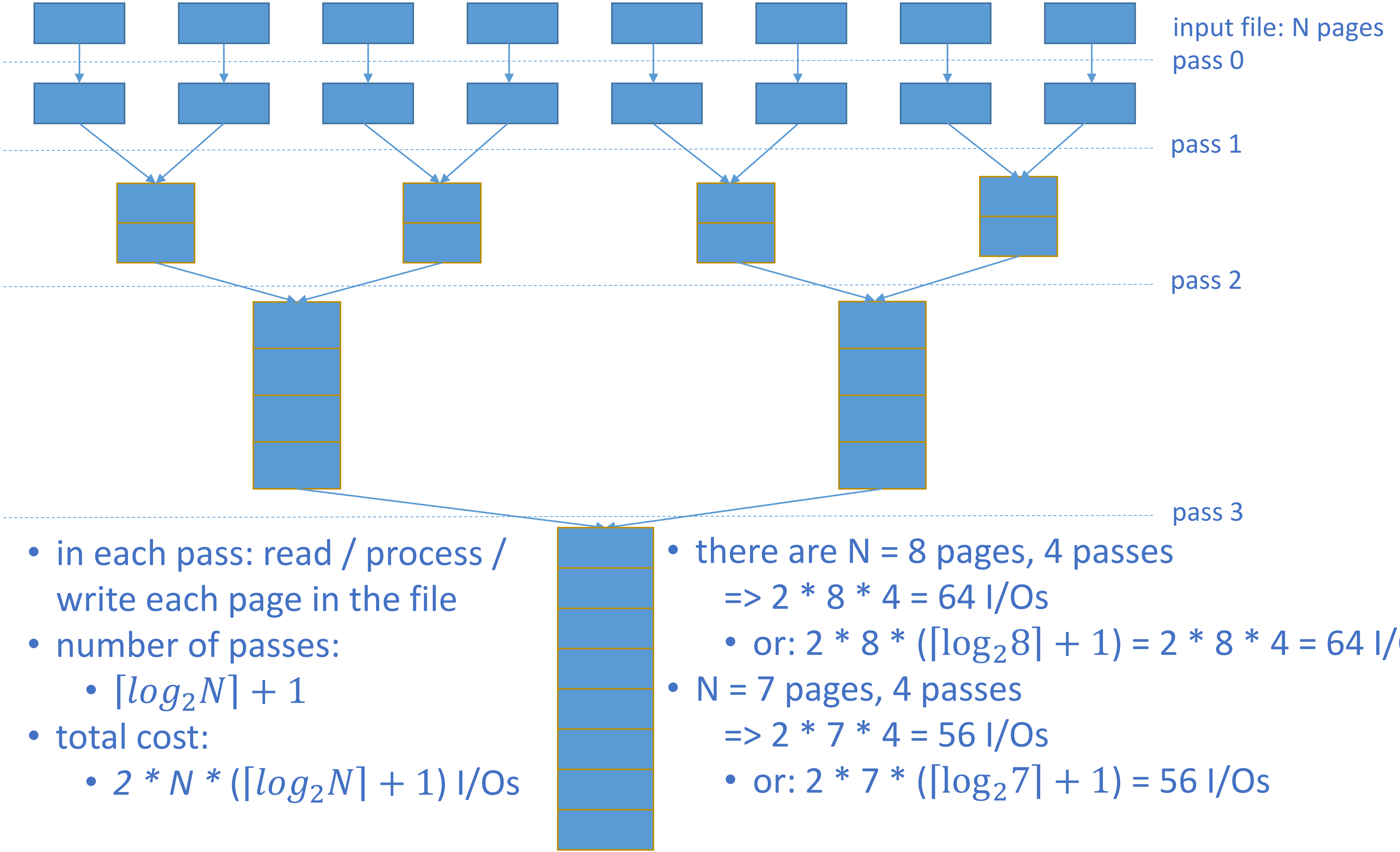
- at each pass, each page in the input file is: read in, processed, and written out; there are 2 I/O operations per page, per pass




- number of passes:
  - $\lceil log_2 N \rceil + 1$

  where *N* is the number of pages in the file to be sorted

- total cost:
  - 2 * number of pages * number of passes
  - *2 * N ** ($\lceil log_2 N \rceil + 1$) I/Os




- in each pass: read / process / write each page in the file
- number of passes:
  - $\lceil log_2 N \rceil + 1$
- total cost:
  - $2 * N * (\lceil log_2 N \rceil + 1)$ I/Os

- there are N = 8 pages, 4 passes
  => 2 * 8 * 4 = 64 I/Os
  - or: $2 * 8 * (\lceil \log_2 8 \rceil + 1) = 2 * 8 * 4 = 64$ I/Os
- N = 7 pages, 4 passes
  => 2 * 7 * 4 = 56 I/Os
  - or: $2 * 7 * (\lceil \log_2 7 \rceil + 1) = 56$ I/Os

## External Merge Sort

- Simple Two-Way Merge Sort: buffer pages are not used effectively
  - for instance, if 200 buffer pages are available, this algorithm still uses only 2 input buffers for passes 1, 2, …
- generalize the Two-Way Merge Sort algorithm to effectively use the available main memory and minimize the number of passes

->

<u>External Merge Sort</u>

- input file to be sorted: N pages
- B buffer pages are available
- <u>pass 0</u>:
  - use B buffer pages
  - read in B pages at a time and sort them in memory

=> $\left\lceil \frac{N}{B} \right\rceil$ runs of B pages each (except for the last one, which may be smaller)

->

# External Merge Sort

- consider again the input file in the previous example:

| 3, 4 | 6, 2 | 9, 4 | 8, 7 | 5, 6 | 3, 1 | 2 |
|---|---|---|---|---|---|---|
| page 1 | page 2 | … | | | | |

- N = 7 (number of pages in the file)
- B = 4 (there are 4 available buffer pages)
- pass 0 produces $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{7}{4} \right\rceil = 2$ runs:
  - use all 4 buffer pages
  - read in 4 pages:

| | | | |
|---|---|---|---|
| 3, 4 | 6, 2 | 9, 4 | 8, 7 |

  - sort the pages in memory, write to disk a run that is 4 pages long:

| | | | |
|---|---|---|---|
| 2, 3 | 4, 4 | 6, 7 | 8, 9 |

  - read in remaining 3 pages:

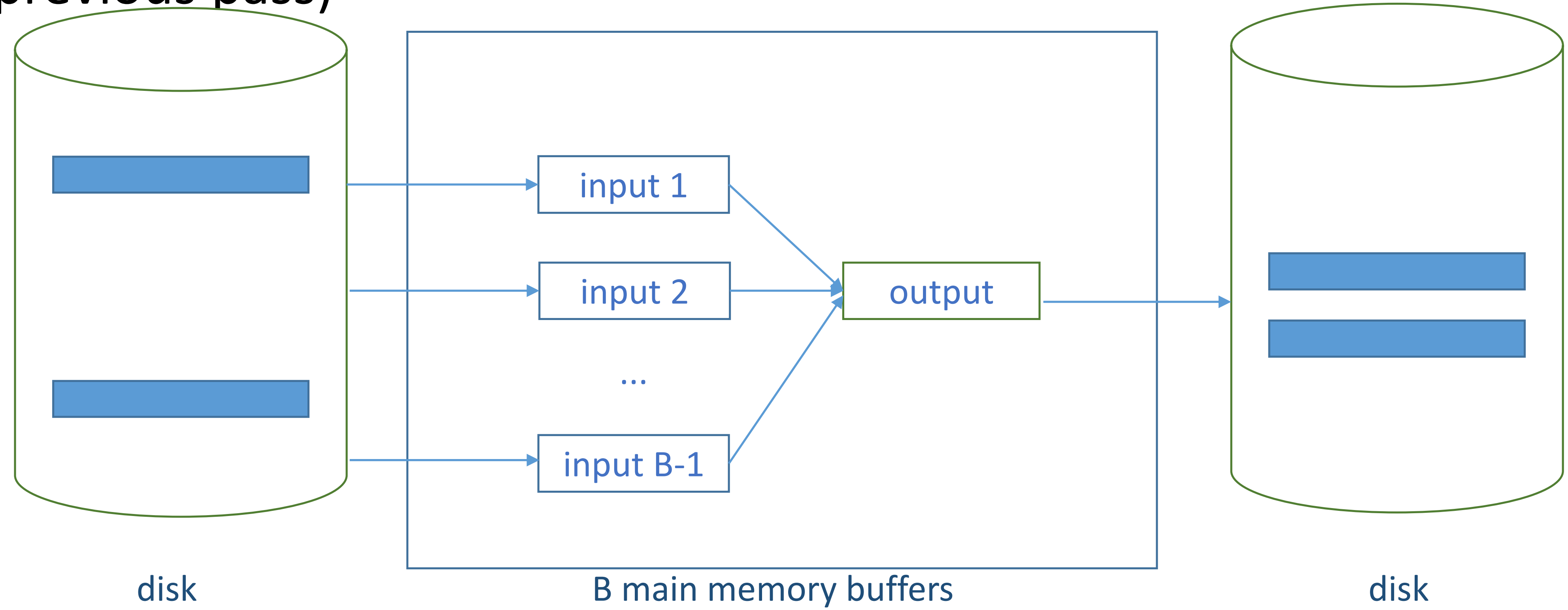
| | | |
|---|---|---|
| 5, 6 | 3, 1 | 2 |

  - sort pages in memory, write to disk run of 3 pages:

| | | |
|---|---|---|
| 1, 2 | 3, 5 | 6 |

# External Merge Sort

- input file to be sorted: N pages
- B buffer pages are available
- pass 1, 2 …:
  - use B-1 pages for input, and one page for output
  - perform a (B-1)-way merge in each pass (i.e., merge B-1 runs from the previous pass)



# External Merge Sort

- runs at the end of pass 0:

| | | | |
|---|---|---|---|
| 2, 3 | 4, 4 | 6, 7 | 8, 9 |

| | | |
|---|---|---|
| 1, 2 | 3, 5 | 6 |

- pass 1
  - read in & merge the first B-1 = 4-1 = 3 runs from pass 0
  - pass 0 produced only 2 runs in this example; read in and merge these 2 runs:

    => run

| | | | | | | |
|---|---|---|---|---|---|---|
| 1, 2 | 2, 3 | 3, 4 | 4, 5 | 6, 6 | 7, 8 | 9 |

## External Merge Sort

- another example:
  - 5 buffer pages B = 5
  - sort file with 108 pages N = 108

- pass 0
  - use all 5 buffer pages
  - read in the first 5 pages of the file, sort them in memory, write the resulting run to disk (5 pages long)
  - read in the next 5 pages of the file, sort them in memory, write the resulting run to disk (5 pages long)
  - …
  - read in the remaining 3 pages of the file, sort them in memory, write the resulting run to disk (3 pages long)
  - 21 runs are 5 pages long; 1 run is 3 pages long

External Merge Sort

- another example: B = 5, N = 108
- pass 0
  - at the end of pass 0 there are $\left\lceil \frac{N}{B} \right\rceil = \left\lceil \frac{108}{5} \right\rceil$ = 22 runs
- pass 1
  - use B-1 = 5-1 = 4 pages for input, and one page for output
  - do a 4-way merge: read in and merge 4 runs from the previous pass
  - read in the first 4 runs from pass 0 (each run into an input buffer)
    - merge the runs and write to the output buffer
    - write the output buffer to disk one page at a time

  => a run that is 20 pages long (4 runs from pass 0 times 5 pages per run)
  - read in the next 4 runs from pass 0; merge the runs and write to the output buffer; write the output buffer to disk one page at a time

  => another run (20 pages long)

  …

# External Merge Sort

- another example: B = 5, N = 108
- pass 0
  - at the end of pass 0 there are 22 runs

- pass 1
  - read in the last 2 runs from pass 0 (one has 5 pages, the other one has 3 pages)
    - merge the runs and write to the output buffer; write the output buffer to disk one page at a time

  => the last run (8 pages long)
  - at the end of pass 1 there are $\left\lceil \frac{22}{4} \right\rceil$ = 6 runs
  - 5 runs are 20 pages long; 1 run is 8 pages long

## External Merge Sort

- another example: B = 5, N = 108
- pass 1
  - at the end of pass 1 there are 6 runs

- pass 2
  - 4-way merge
  - read in the first 4 runs from pass 1
    - merge the runs and write to the output buffer; write the output buffer to disk one page at a time

    => a run that is 80 pages long (4 runs from pass 1 times 20 pages per run)
  - read in the remaining 2 runs from pass 1 (20 and 8 pages, respectively)
    => a run that is 28 pages long
  - at the end of pass 2 there are $\left\lceil \frac{6}{4} \right\rceil$ = 2 runs

## External Merge Sort

- another example: B = 5, N = 108
- pass 2
  - at the end of pass 2 there are 2 runs

- pass 3
  - read in the 2 runs from pass 2 and merge them
    => a run that is 108 pages long, representing the sorted file

# External Merge Sort

- cost
  - N – number of pages in the input file, B – number of available pages in the buffer
  - at each pass: read / process / write each page
  - number of passes: $\lceil log_{B-1}\lceil N/B \rceil \rceil + 1$
  - total cost: $2 * N * \left( \left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1 \right)$ I/Os

- previous example: B = 5 and N = 108, with 4 passes over the data
  - cost:
    2 * 108 * 4 = 864 I/Os
  - $2 * 108 * \left( \left\lceil log_{5-1} \left\lceil \frac{108}{5} \right\rceil \right\rceil + 1 \right) = 216 * (\lceil log_4 22 \rceil + 1) =$
    $216 * 4 = 864$ I/Os

- B buffer pages
- sort file with N pages

<u>Simple Two-Way Merge Sort</u>

pass 0 => N runs

number of passes = $\lceil log_2 N \rceil + 1$

<u>External Merge Sort</u>

pass 0 => $\left\lceil \frac{N}{B} \right\rceil$ runs

number of passes = $\left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1$

- External Merge Sort – reduced number of:
    - runs produces by the 1st pass
    - passes over the data
- B is usually large => significant performance gains

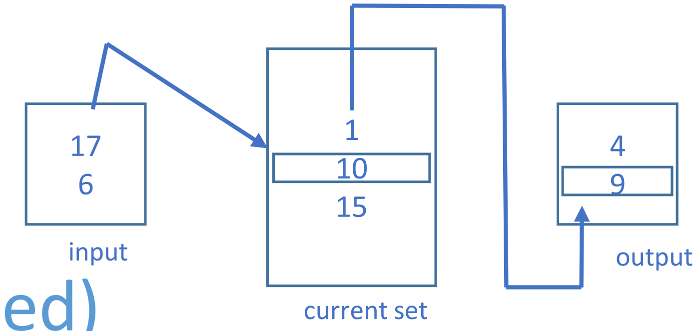
# <u>External Merge Sort</u> – number of passes for different values of N and B

| 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 |

* <u>minimize the number of runs</u> - optional

- external merge sort
  - N pages in the file, B buffer pages => $\lceil N/B \rceil$ runs of B pages each
- <u>improvement</u>
  - algorithm known to produce sorted <u>runs</u> of approximately <u>2*B pages</u> (on average)
  - use 1 page as an input buffer, 1 page as an output buffer
  - the remaining buffer pages are collectively referred to as the *current set*

- <u>example</u> - sort file in ascending order on some key *k*:
- repeatedly pick record *r* with smallest *k* in current set such that *k* is > the largest *k* in the output buffer
- append *r* to output buffer (the output buffer is kept sorted)
- use the extra space in the current set to bring in the next tuple from the input buffer



* minimize the number of runs - optional

- process all tuples in the input buffer, then read in the next page of the file
- when the output buffer fills up, write it to disk (add its content to the run that is currently being built)
- the current run is completed when every *k* value in the current set is < the largest *k* value in the output buffer; when this happens, the output buffer is written out (its content becomes the last page in the current run), and a new run is started

## Sort-Merge Join

- equality join, one join column: $\mathrm{E} \otimes_{i=j} S$ ($i^{th}$ column's value in E = $j^{th}$ column's value in S)
- sort E and S on the join column (if not already sorted):
  - for instance, by using External Merge Sort

  => *partitions* = groups of tuples with the same value in the join column
- merge E and S; look for tuples *e* in E, *s* in S such that $e_i = s_j$:
  - while *current* $e_i <$ *current* $s_j$
    - advance the scan of E
  - while *current* $e_i >$ *current* $s_j$
    - advance the scan of S
  - if *current* $e_i =$ *current* $s_j$
    - output joined tuples <*e, s*>, where *e* and *s* are in the current partition (i.e., they have the same value in the $i^{th}$ and $j^{th}$ column, respectively)
    - there could be multiple tuples in E with the same value in the $i^{th}$ column as the current tuple *e* (same is true for S)

# Sort-Merge Join

- partitions are illustrated on tables Students and Exams below (join column SID in both tables):

| SID | SName | Age |
|---|---|---|
| 20 | Ana | 20 |
| 30 | Dana | 20 |
| 40 | Dan | 20 |
| 45 | Daniel | 20 |
| 50 | Ina | 20 |

| SID | CID | EDate | Grade | FacultyMember |
|---|---|---|---|---|
| 30 | 2 | 20/1/2018 | 10 | Ionescu |
| 30 | 1 | 21/1/2018 | 9.99 | Pop |
| 45 | 2 | 20/1/2018 | 9.98 | Ionescu |
| 45 | 1 | 21/1/2018 | 9.98 | Pop |
| 45 | 3 | 22/1/2018 | 10 | Stan |
| 50 | 2 | 20/1/2018 | 10 | Ionescu |

# Sort-Merge Join

- during the merging phase, E is scanned once; every partition in S is scanned as many times as there are matching tuples in the corresponding partition in E

E

| … | $i^{th}$ column | … |
|---|---|---|
| | 1 | |
| | 2 | |
| | 2 | |
| | 2 | |
| | 3 | |
| | … | |

S

| … | $j^{th}$ column | … |
|---|---|---|
| | 0 | |
| | 2 | partition P |
| | 2 | |
| | 4 | |
| | … | |
| | … | |

  - for instance, partition P in the above table S is scanned 3 times, once per matching tuple in the corresponding partition in E
  - there are 6 output joined tuples <*e, s*> for partition P
- this algorithm avoids the enumeration of the cross-product: tuples in a partition in E are compared only with the S tuples in the same partition!

# Sort-Merge Join

- cost:
  - sorting E
    - cost: O(MlogM)
  - sorting S
    - cost: O(NlogN)
  - cost of merging: M + N I/Os, assuming partitions in S are scanned only once
    - worst-case scenario: M * N I/Os (when all records in E and S have the same value in the join column)

* E - M pages; S - N pages*

Sort-Merge Join (Exams$\otimes_{Exams.SID=Students.SID}$ Students)

- 100 buffer pages
  - sort Exams
    - 2 passes => cost: 2 * 2 * 1000 = 4000 I/Os
  - sort Students
    - 2 passes => cost: 2 * 2 * 500 = 2000 I/Os
  - merging phase
    - cost: 1000 + 500 = 1500 I/Os
  - total cost: 4000 + 2000 + 1500 = 7500 I/Os
    - similar to the cost of Block Nested Loops Join

| | | |
|---|---|---|
| * E - M pages, $p_E$ records / page * | * 1000 pages * | * 100 records / page* |
| * S - N pages, $p_S$ records / page * | * 500 pages * | * 80 records / page * |

Sort-Merge Join (Exams$\otimes_{Exams.SID=Students.SID}$ Students)

- 35 buffer pages
  - sort Exams
    - 2 passes => cost: 2 * 2 * 1000 = 4000 I/Os
  - sort Students
    - 2 passes => cost: 2 * 2 * 500 = 2000 I/Os
  - merging phase
    - cost: 1000 + 500 = 1500 I/Os
  - total cost: 4000 + 2000 + 1500 = 7500 I/Os
    - ex: compute cost of BNLJ and compare

* E - M pages, $p_E$ records / page *    * 1000 pages *    * 100 records / page*

* S - N pages, $p_S$ records / page *    * 500 pages *    * 80 records / page *

Sort-Merge Join (Exams$\otimes_{Exams.SID=Students.SID}$ Students)

- 300 buffer pages
  - sort Exams
    - 2 passes => cost: 2 * 2 * 1000 = 4000 I/Os
  - sort Students
    - 2 passes => cost: 2 * 2 * 500 = 2000 I/Os
  - merging phase
    - cost: 1000 + 500 = 1500 I/Os
  - total cost: 4000 + 2000 + 1500 = 7500 I/Os
    - ex: compute cost of BNLJ and compare

* E - M pages, $p_E$ records / page * * 1000 pages * * 100 records / page*

* S - N pages, $p_S$ records / page * * 500 pages * * 80 records / page *

# References


- [Ra00] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (2nd Edition), McGraw-Hill, 2000
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga08] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book, Prentice Hall Press, 2008
- [Ra07] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si10] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, McGraw-Hill, 2010, http://codex.cs.yale.edu/avi/db-book/
- [Ul11] ULLMAN, J., WIDOM, J., A First Course in Database Systems, http://infolab.stanford.edu/~ullman/fcdb.html