Database Management Systems

Lecture 8

Evaluating Relational Operators

Query Optimization (II)

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

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

- 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

Sorting

- explicitly required
  - SELECT ... ORDER BY list
- used by operators like:
  - duplicate elimination
  - join
  - union
  - intersection
  - set-difference
  - grouping

Sorting

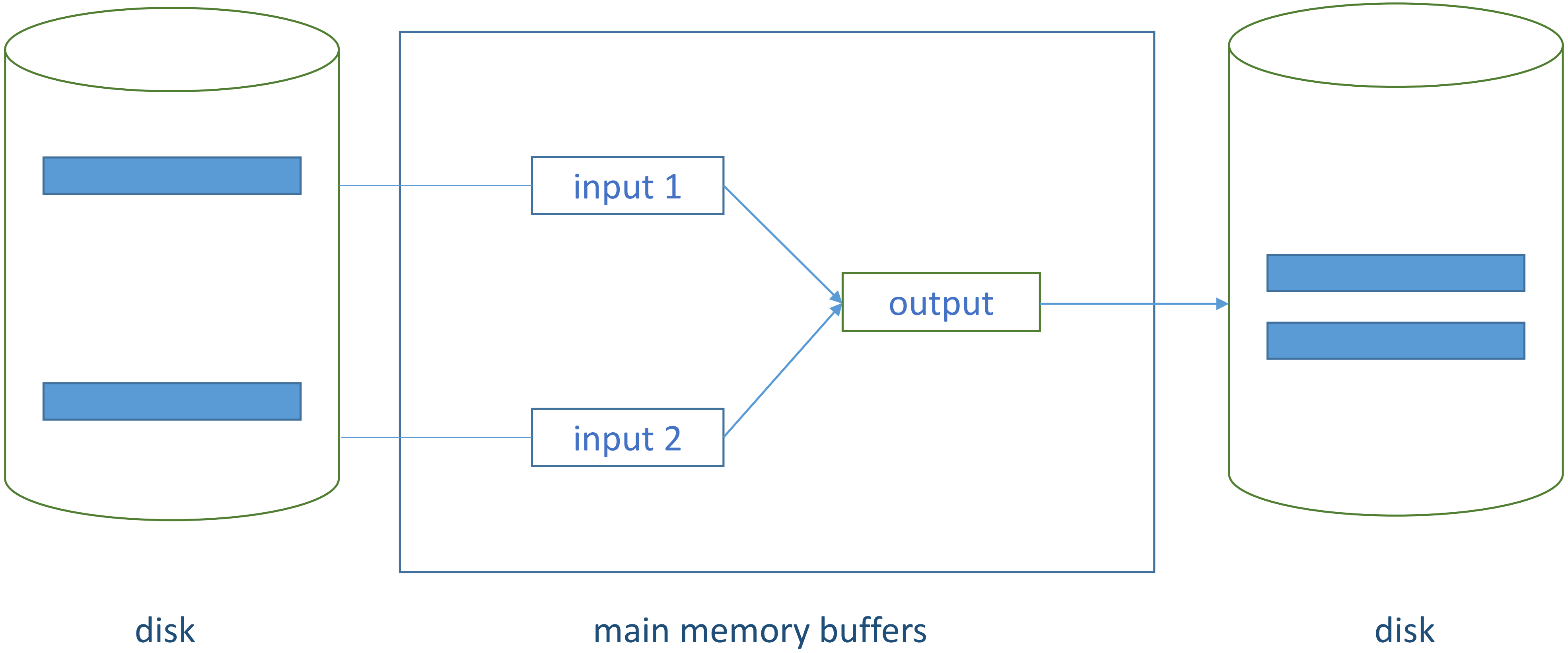
- v1
  - data fits into available main memory

  => use an internal sorting algorithm
- v2
  - data doesn't fit into available main memory

  => use an external sorting algorithm
    - objective
      - minimize cost of disk accesses
    - create *runs*
      - sort records in the data source that fit into main memory
    - place runs into temporary files
    - merge runs using an external sorting algorithm

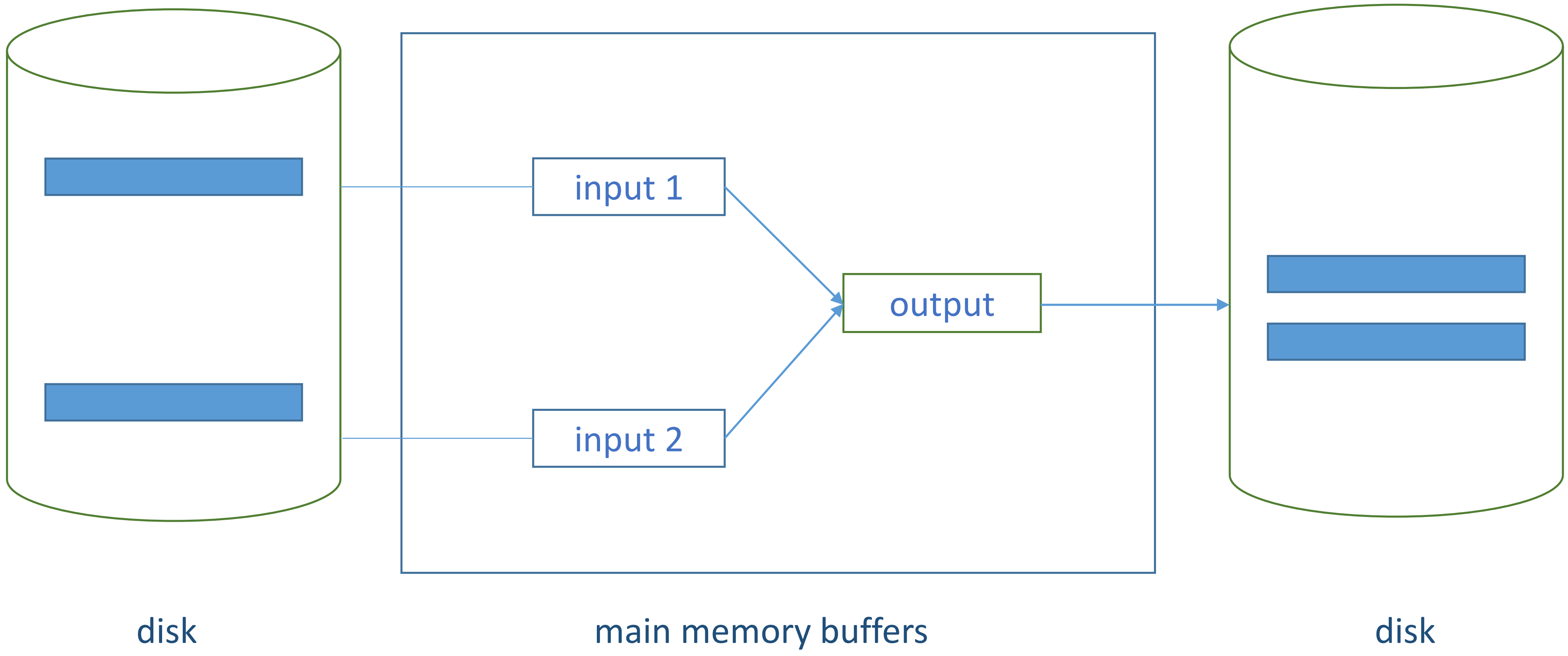
Simple Two-Way Merge Sort

- 3 buffer pages
- repeated passes over the data
- even large data sources can be sorted with a small amount of main memory



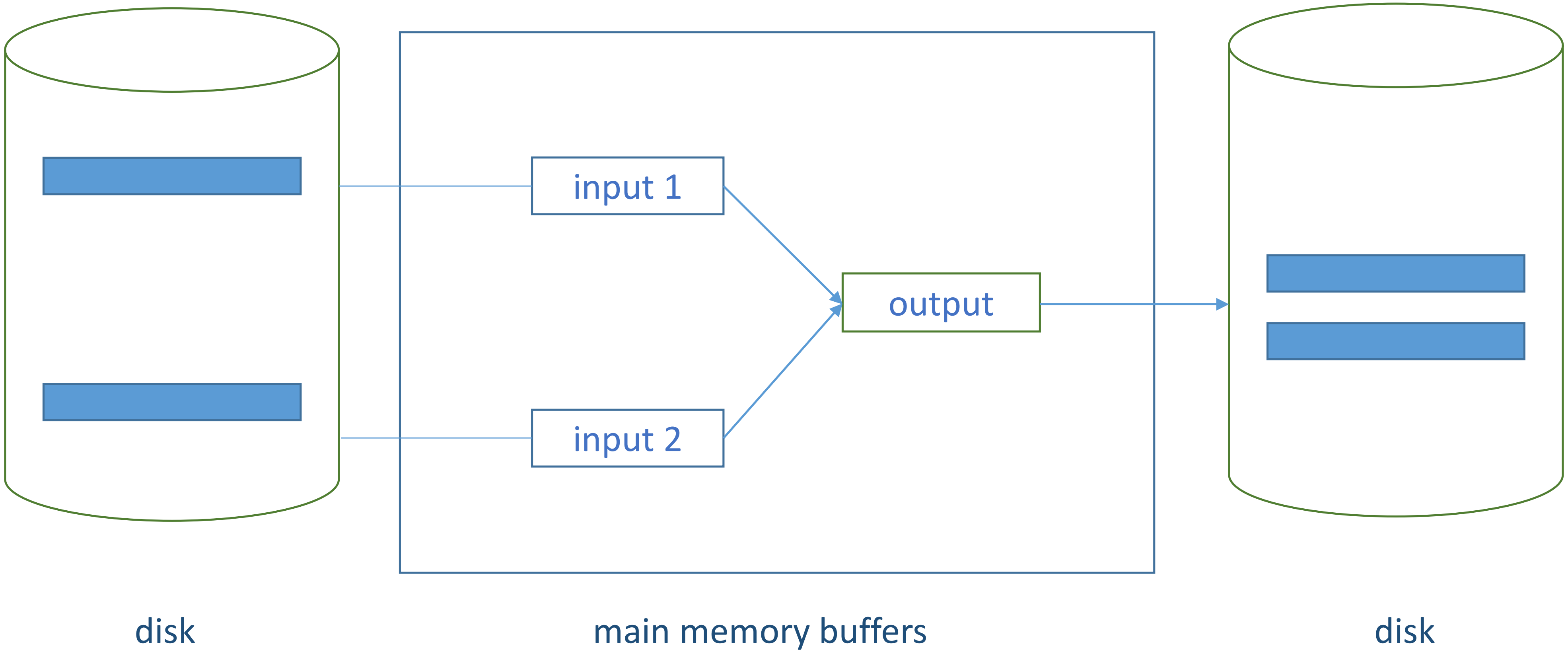
Simple Two-Way Merge Sort

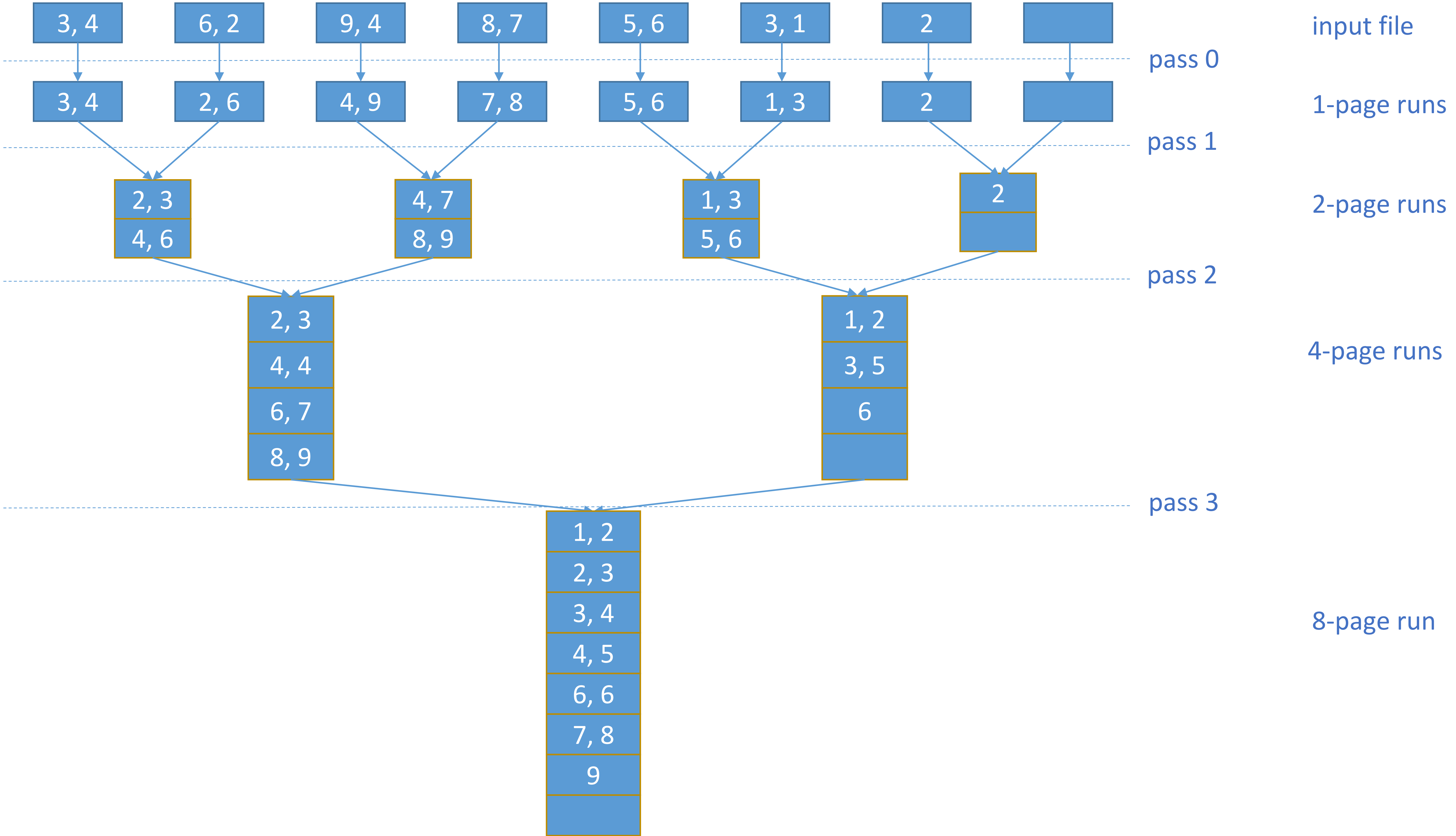
- pass 0
  - read page -> sort page -> save page
    => 1-page runs
  - uses one buffer page

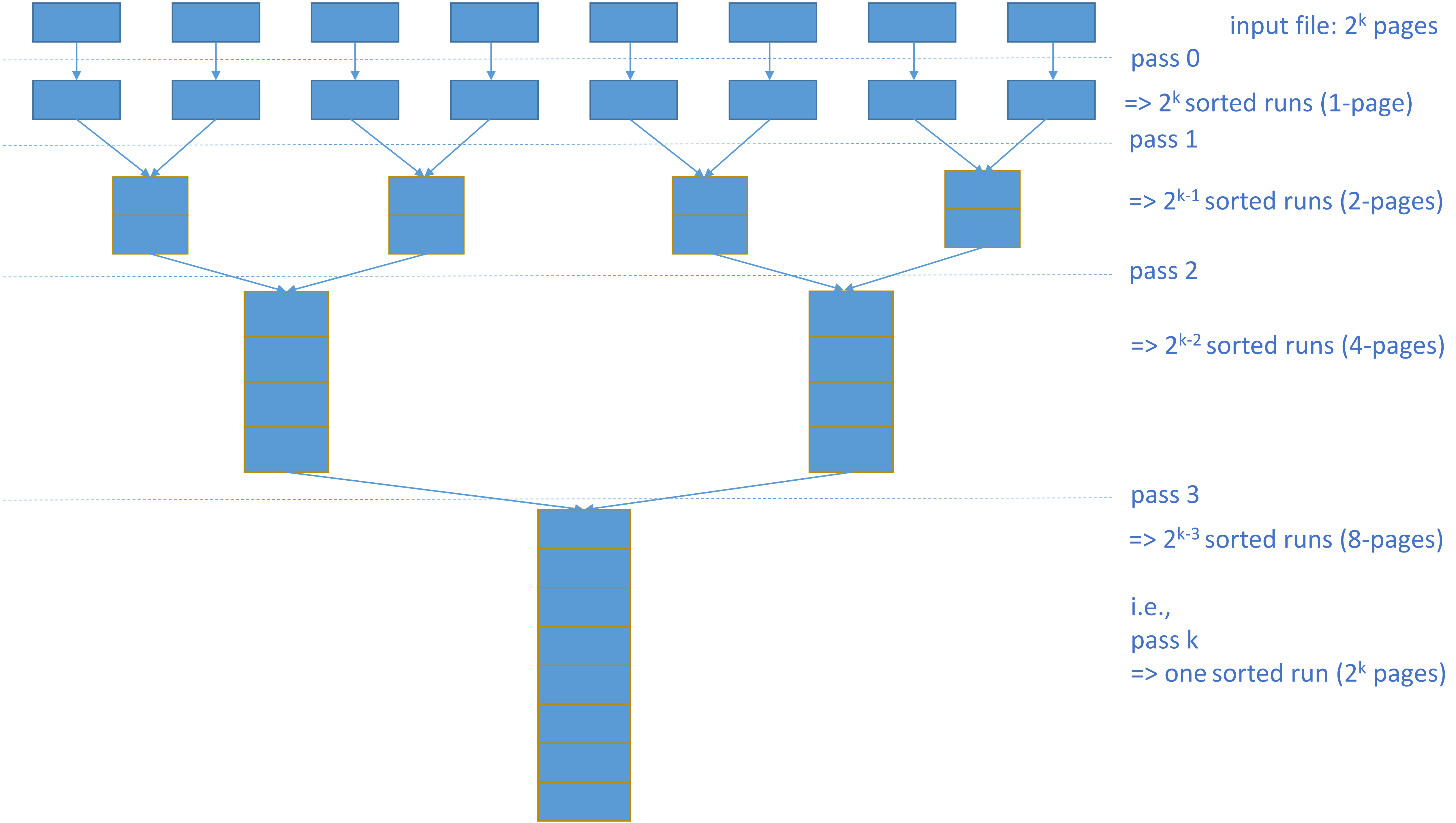


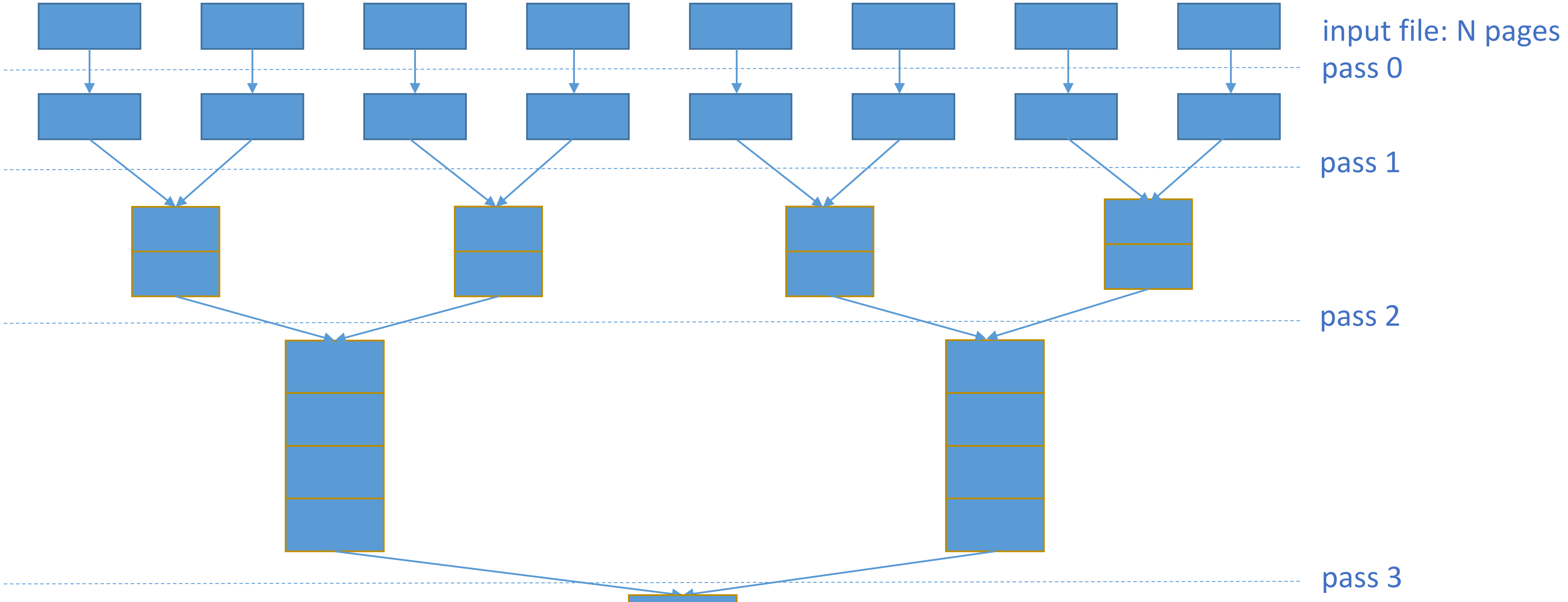
Simple Two-Way Merge Sort

- 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




input file
3, 4
6, 2
9, 4
8, 7
5, 6
3, 1
2
pass 0
1-page runs
3, 4
2, 6
4, 9
7, 8
5, 6
1, 3
2
pass 1
2-page runs
2, 3
4, 6
4, 7
8, 9
1, 3
5, 6
2
pass 2
4-page runs
2, 3
4, 4
6, 7
8, 9
1, 2
3, 5
6
pass 3
8-page run
1, 2
2, 3
3, 4
4, 5
6, 6
7, 8
9


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)

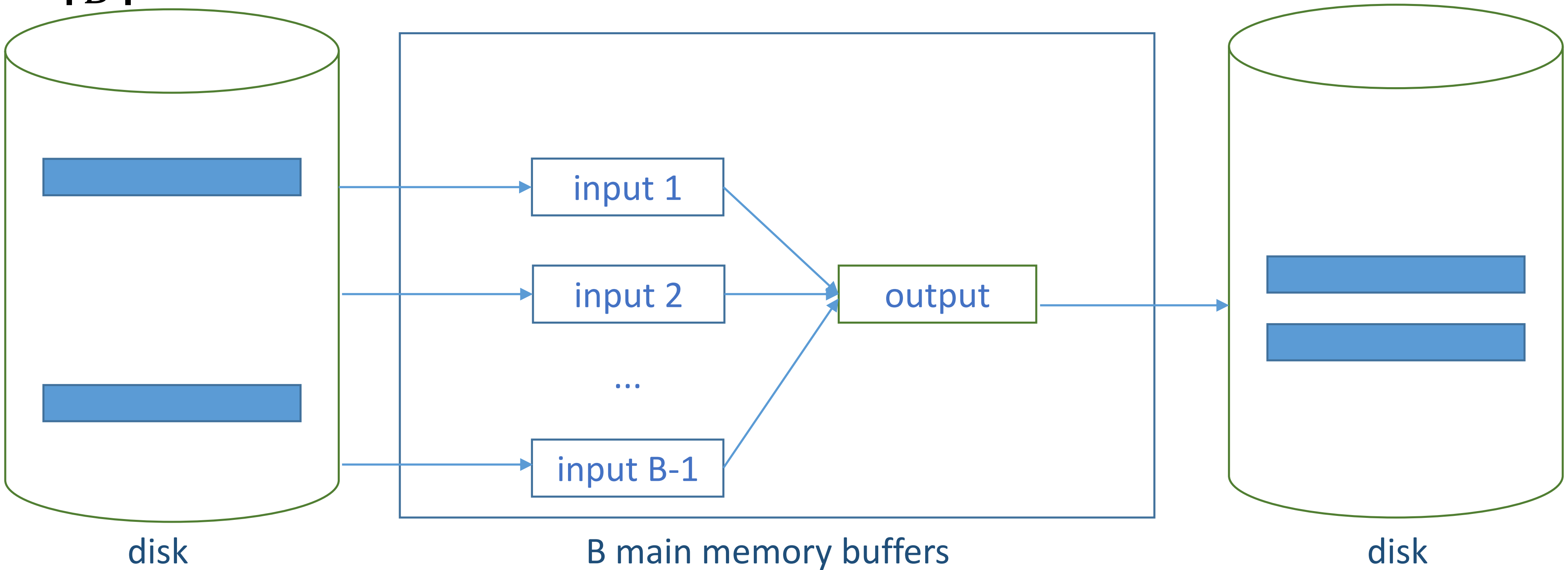


- each pass:
  - read / process / write each page
- number of passes:
  - $\lceil log_2 N \rceil + 1$
- total cost:
  - $2 * N * (\lceil log_2 N \rceil + 1)$ I/Os
- N = 8
  - 2 * 8 * 4 = 64 I/Os
  - $2 * 8 * (\lceil \log_2 8 \rceil + 1)$
    - 2 * 8 * 4 = 64 I/Os
- N = 7
  - 2 * 7 * 4 = 56 I/Os
  - $2 * 7 * (\lceil \log_2 7 \rceil + 1)$ = 56 I/Os

External Merge Sort

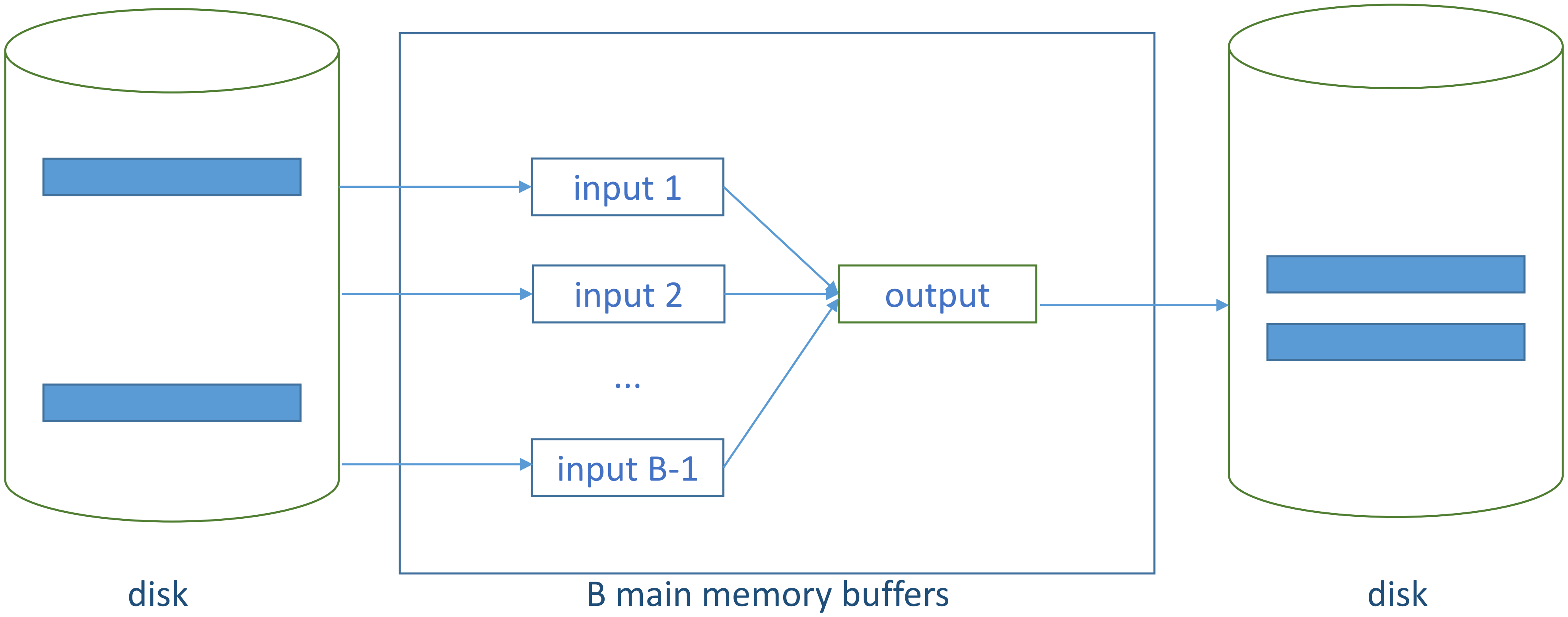
- Simple Two-Way Merge Sort – buffer pages are not used effectively
- generalize algorithm – minimize the number of passes
  - pass 0
    - use B buffer pages
    - read in B pages at a time and sort them

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



External Merge Sort

- generalize algorithm
  - pass 1, 2 ...
    - use B-1 pages for input, one page for output
    - (B-1)-way merge in each pass (i.e., merge B-1 runs from the previous pass)



External Merge Sort

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

- pass 0
  => $\left\lceil \frac{108}{5} \right\rceil$ = 22 sorted runs
  - 21 runs – 5 pages long
  - 1 run – 3 pages long
- pass 1
  => $\left\lceil \frac{22}{4} \right\rceil$ = 6 sorted runs
  - 5 runs – 20 pages long
  - 1 run – 8 pages long

External Merge Sort

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

- pass 2
  => $\left\lceil \frac{6}{4} \right\rceil$ = 2 sorted runs
  - 80 pages long
  - 28 pages long
- pass 3
  => 1 sorted run – 108 pages

External Merge Sort

- cost
  - 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
- e.g., B = 5 and N = 108
  - 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 = $\left\lceil log_{B-1} \left\lceil \frac{N}{B} \right\rceil \right\rceil + 1$

- B is usually large => significant performance gains
- can use various algorithms to sort data in pages (e.g., Quick Sort)

- External Merge Sort – number of passes

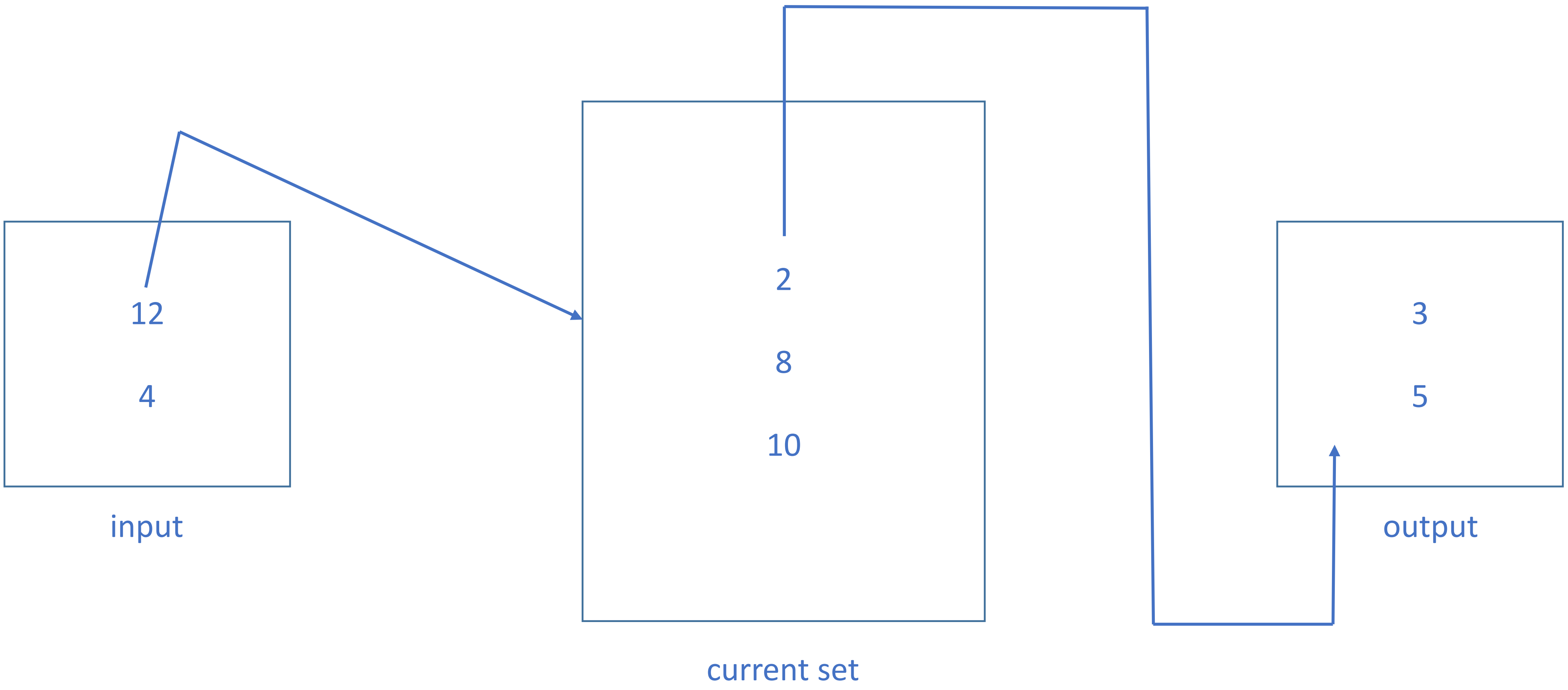
N	B = 3	B = 5	B = 9	B = 17	B = 129	B = 257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

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

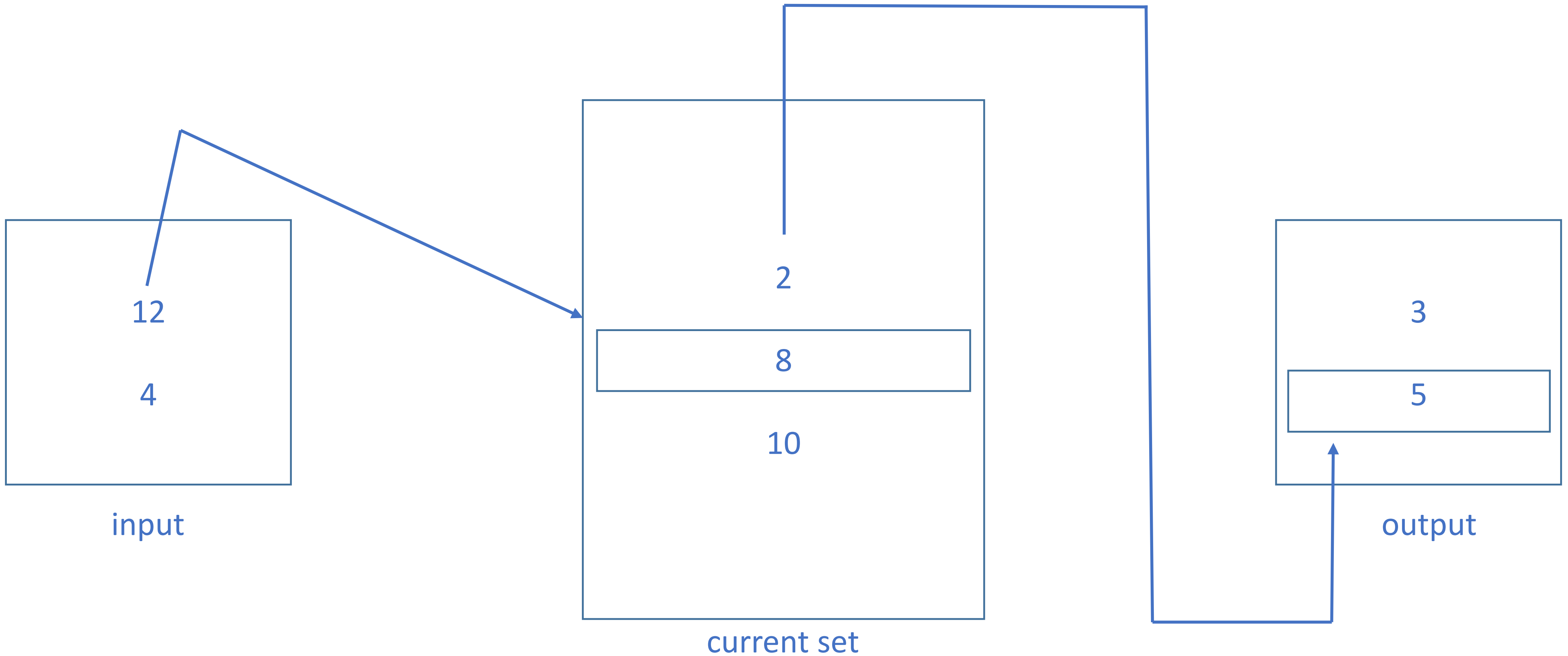
    - sort file in ascending order on some *k*

->

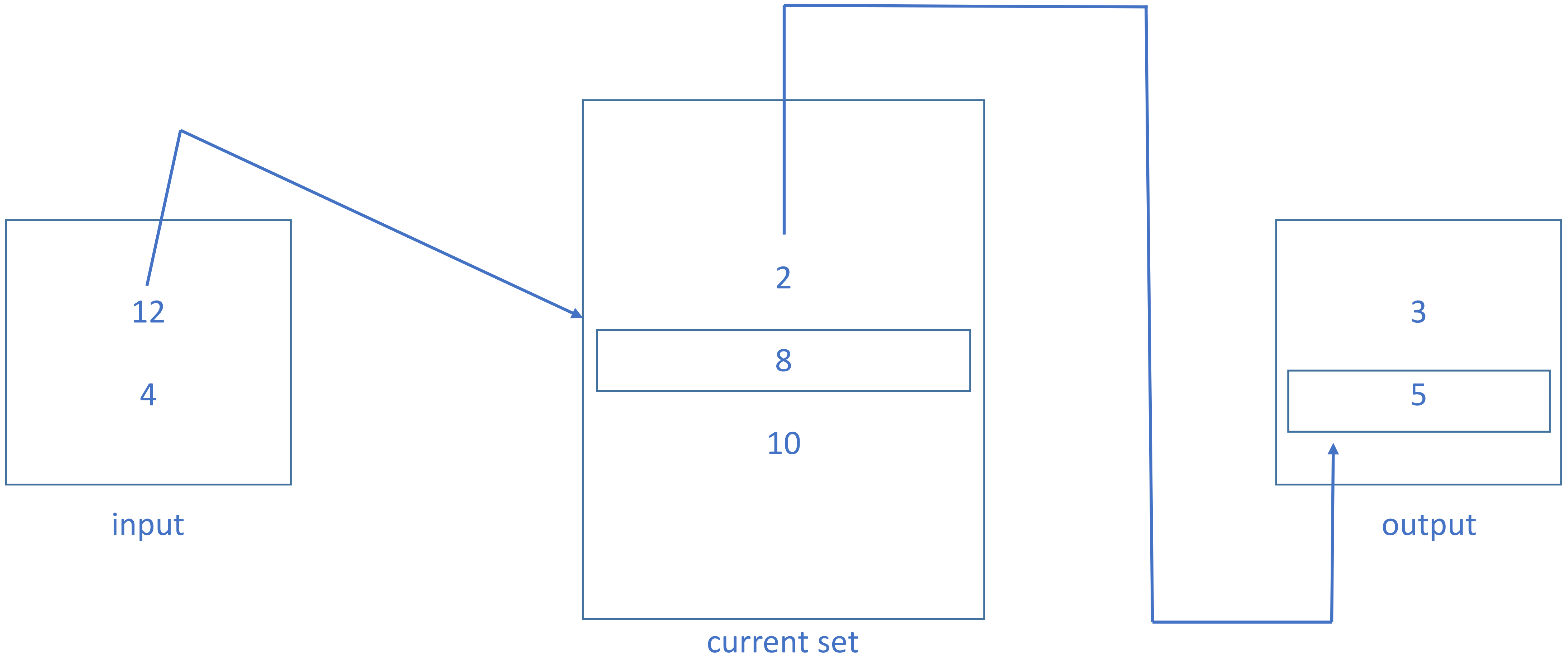
- minimize the number of runs



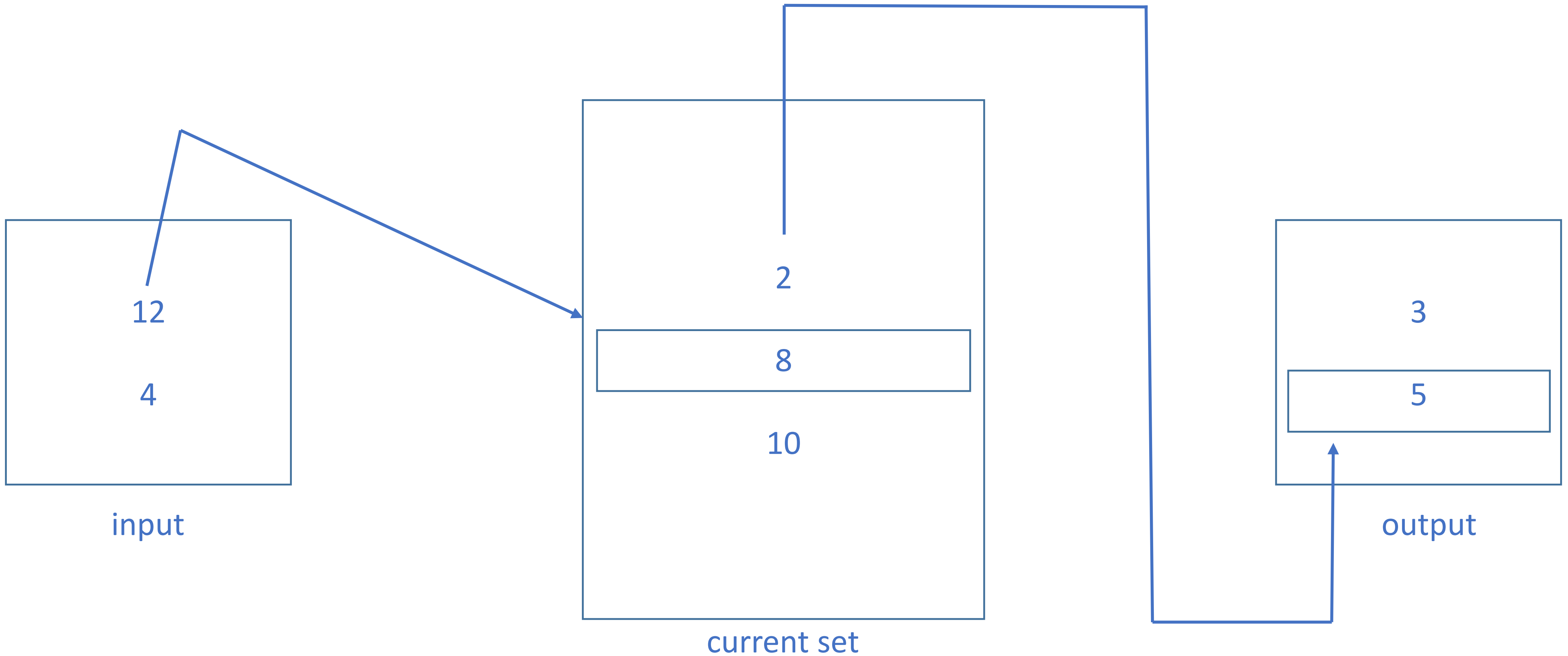
- minimize the number of runs
  - repeatedly pick tuple t with smallest k in current set such that k is greater than the largest k in the output buffer
  - append t to output buffer (the output buffer is kept sorted)
  - use extra space in current set to bring in next tuple from input buffer



- minimize the number of runs
  - all tuples in input buffer consumed
    => read in next page from file
  - output buffer full
    - write it out, extend current run



- minimize the number of runs
  - current run completed when every *k* value in current set is smaller than the largest *k* value in the output buffer
    - write out output buffer (i.e., last page in current run)
    - start new run



Sort-Merge Join ($\mathrm{E} \otimes_{i=j} \mathrm{S}$)

- sort E and S on the join column
  - e.g., external sorting

  => *partitions*, i.e., groups of tuples with the same value in the join column
- merge E and S, return qualifying tuples <e, s>
  - avoid enumeration of the cross-product
    - compare E tuples in a partition with only the S tuples in the same partition
  - 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 e_i, s_j)

Sort-Merge Join ($\mathrm{E} \otimes_{i=j} \mathrm{S}$)

- merge E and S, return qualifying tuples <e, s>
  - E is scanned once
  - partition P in S is scanned t times, where t is the number of matching tuples in the corresponding partition in E

Sort-Merge Join

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

- cost
  - E is scanned once
  - every partition in S is scanned once for every matching tuple in the corresponding partition in E
  - sorting E
    - cost: O(MlogM)
  - sorting S
    - cost: O(NlogN)
  - cost of merging: M + N I/Os
    - worst-case scenario: M * N I/Os

* 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)

- 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 BNLJ cost: 6000 I/Os

* 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)

- <u>35 buffer pages</u>
  - 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
    - BNLJ cost: 1000 + $\lceil 1000/35 \rceil$ * 500 = 1000 + 29 * 500 = 15.500 I/Os

* 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
    - BNLJ cost = 1000 + $\lceil 1000/300 \rceil$ * 500 = 1000 + 4 * 500 = 3000 I/Os

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

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

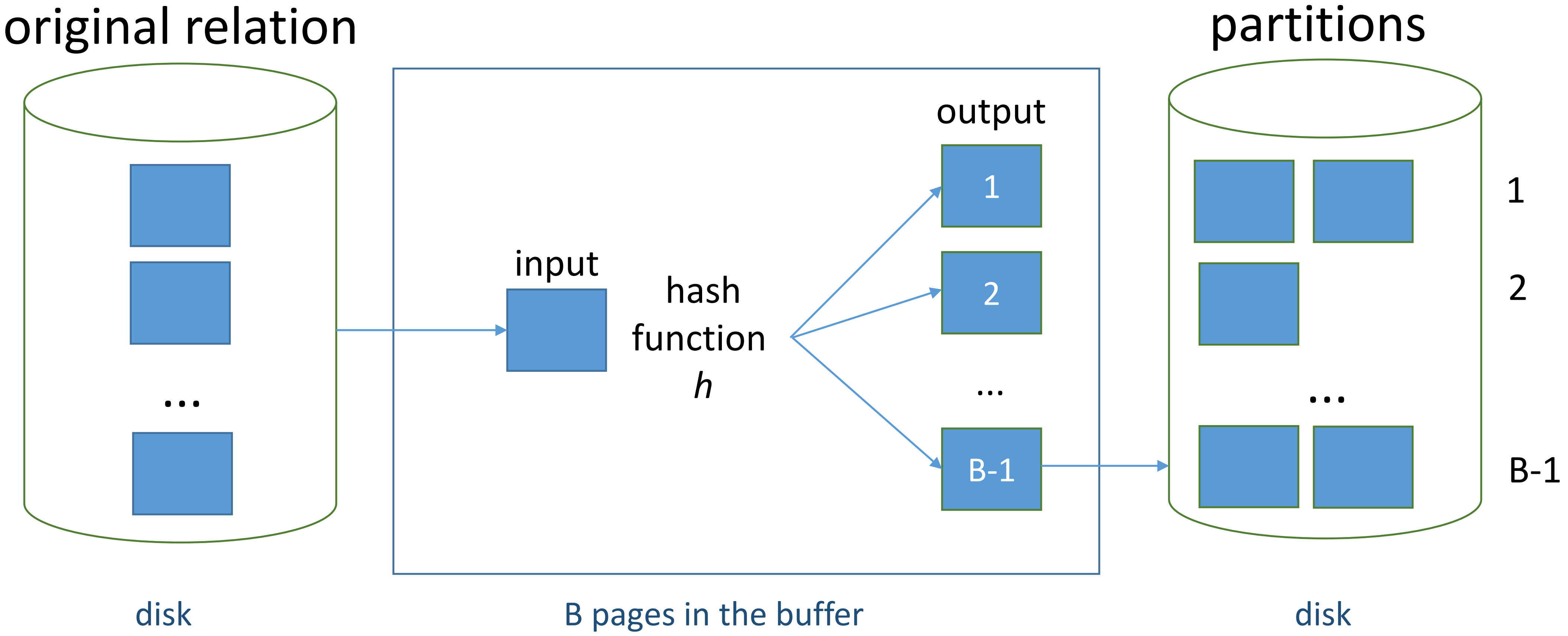
Hash Join (E $\otimes_{i=j}$ S)

- partitioning phase (building phase)
- probing phase (matching phase)

->

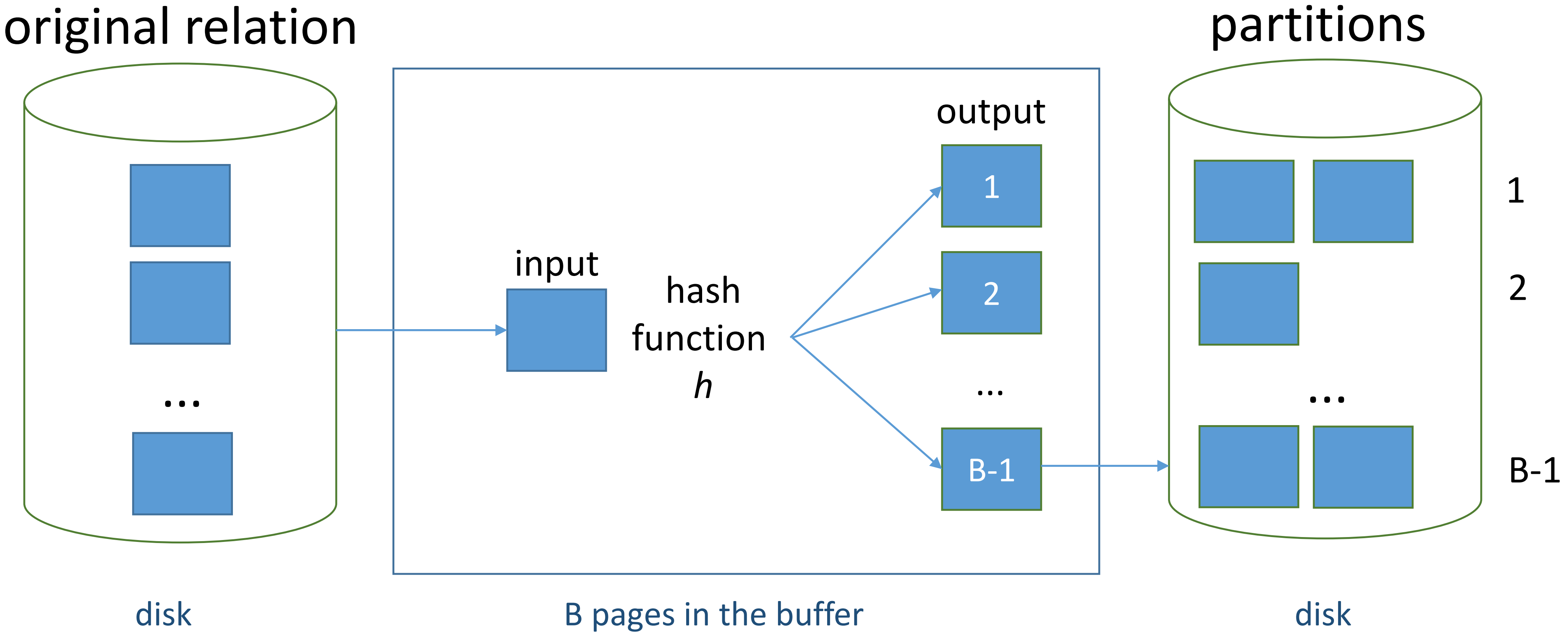
Hash Join ($E \otimes_{i=j} S$)

- partitioning phase
  - hash E and S on the join column with the same hash function *h*
    => partitions
  - choice of *h* - tuples are distributed uniformly to one of B-1 partitions



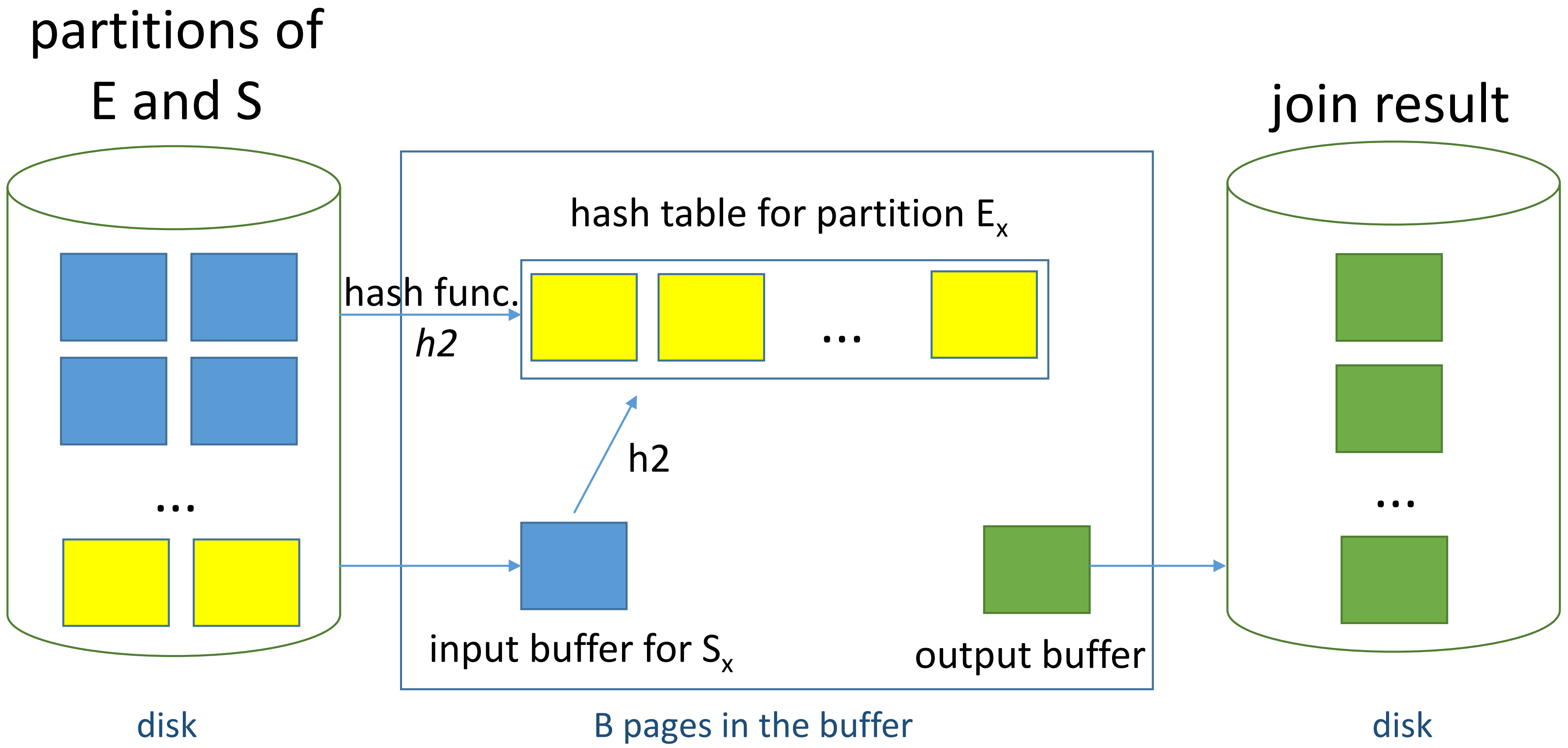
Hash Join ($E \otimes_{i=j} S$)

- partitioning phase
  - *partition*
    - collection of tuples with a common hash value



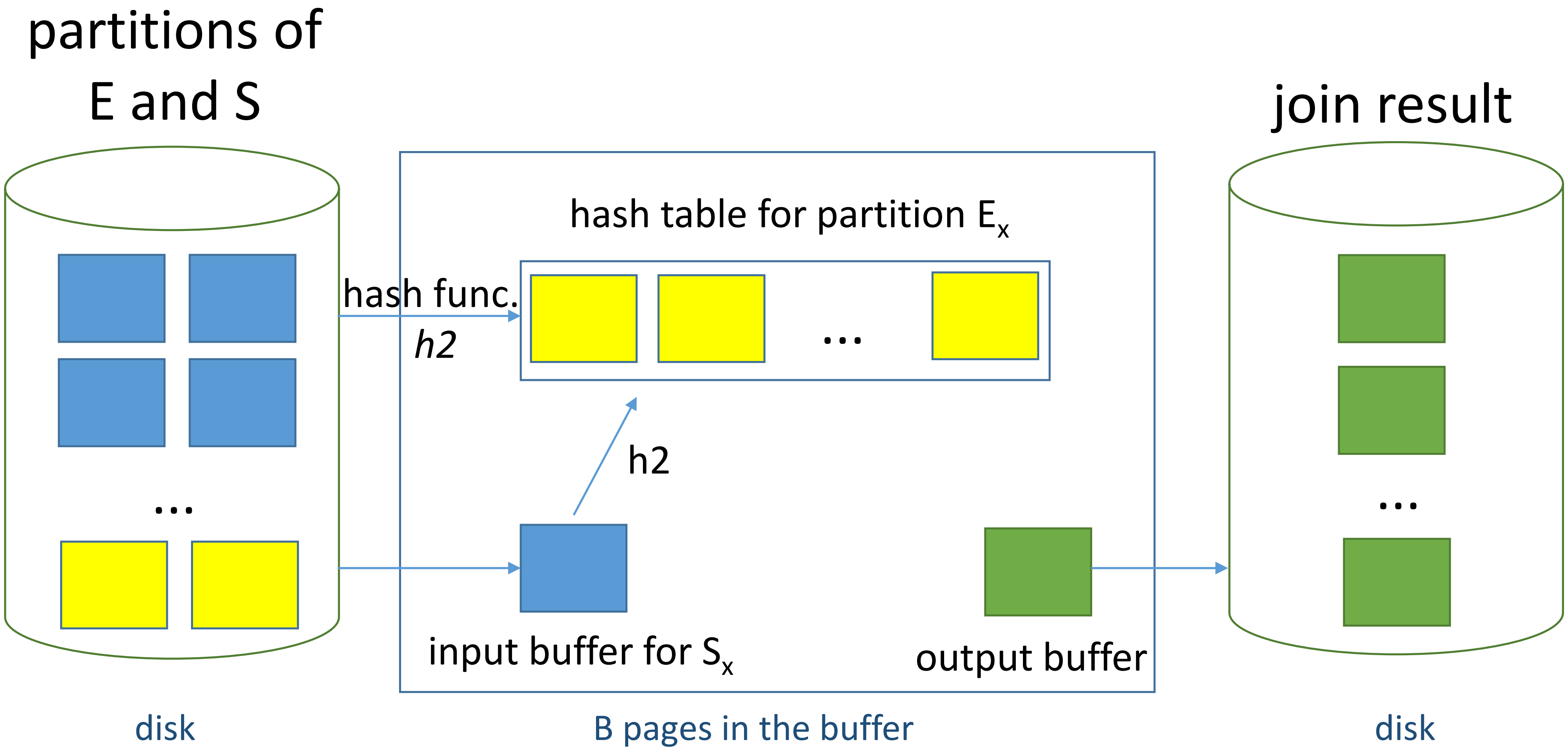
Hash Join ($E \otimes_{i=j} S$)

- probing phase
  - read in a partition of the smaller relation (e.g., E) and scan the corresponding partition of S for matching tuples



Hash Join ($\mathrm{E} \otimes_{i=j} \mathrm{S}$)

- probing phase
  - in practice - build in-memory hash table for the E partition using a different function *h2* (to reduce CPU costs)



Hash Join ($E \otimes_{i=j} S$)

- cost
  - partitioning
    - E, S - read, written once => cost: 2*(M+N) I/Os
  - probing
    - scan each partition once => cost: M+N I/Os

  => total cost: 3*(M+N) I/Os
    - assumption: each partition fits into memory during probing
  - 3*(1000 + 500) = 4500 I/Os

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

Hash Join ($\mathrm{E} \otimes_{i=j} \mathrm{S}$)

- memory requirements
  - objective: partition in E fits into main memory (S - similarly)
    - B buffer pages
    - need one input buffer

      => maximum number of partitions: B-1
    - size of largest partition: B - 2
      - need one input buffer for S, one output buffer
    - assume uniformly sized partitions

      => size of each E partition: M/(B-1)

    => M/(B-1) < B-2 => we need approximately B > $\sqrt{\mathrm{M}}$
    - in-memory hash table to speed up tuple matching => need a little more memory

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

Hash Join ($\mathrm{E} \otimes_{i=j} \mathrm{S}$)

- memory requirements
  - if h does not partition E uniformly, some E partitions may not fit in memory during probing phase – *partition overflow*
    => apply hash join technique recursively to do the join of an overflowing E partition with the corresponding S partition
    - divide E, S into subpartitions
    - join subpartitions pairwise
    - if subpartitions don't fit in memory, apply hash join technique recursively

general join conditions

- equalities over several attributes
  - E.SID = S.SID AND E.attrE = S.attrS
    - index nested loops join
      - Exams – inner relation
        - build index on Exams with search key <SID, attrE> (if not already created)
        - can also use index on SID or index on attrE
      - Students – inner relation (similar)
    - sort-merge join
      - sort Exams on <SID, attrE>, sort Students on <SID, attrS>
    - hash join
      - partition Exams on <SID, attrE>, partition Students on <SID, attrS>
    - other join algorithms
      - essentially unaffected

general join conditions

- inequality comparison
    - E.attrE < S.attrS
        - index nested loops join
            - B+ tree index required
        - sort-merge join
            - not applicable
        - hash join
            - not applicable
        - other join algorithms
            - essentially unaffected

* no join algorithm is uniformly superior to others

- choice of a good algorithm depends on:
  - sizes of:
    - joined relations
    - buffer pool
  - available access methods

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