Database Management Systems

Lecture 7

Evaluating Relational Operators

Query Optimization

Sabina S. CS

* queries – composed of <u>relational operators</u>:

- selection (σ)
  - selects a subset of records from a relation
- projection (π)
  - eliminates certain columns from a relation
- join ($\otimes$)
  - combines data from two relations
- cross-product ($R1 \times R2$)
  - returns every record in R1 concatenated with every record in R2
- set-difference ($R1 - R2$)
  - returns records that belong to R1 and don’t belong to R2
- union ($R1 \cup R2$)
  - returns all records in relations R1 and R2

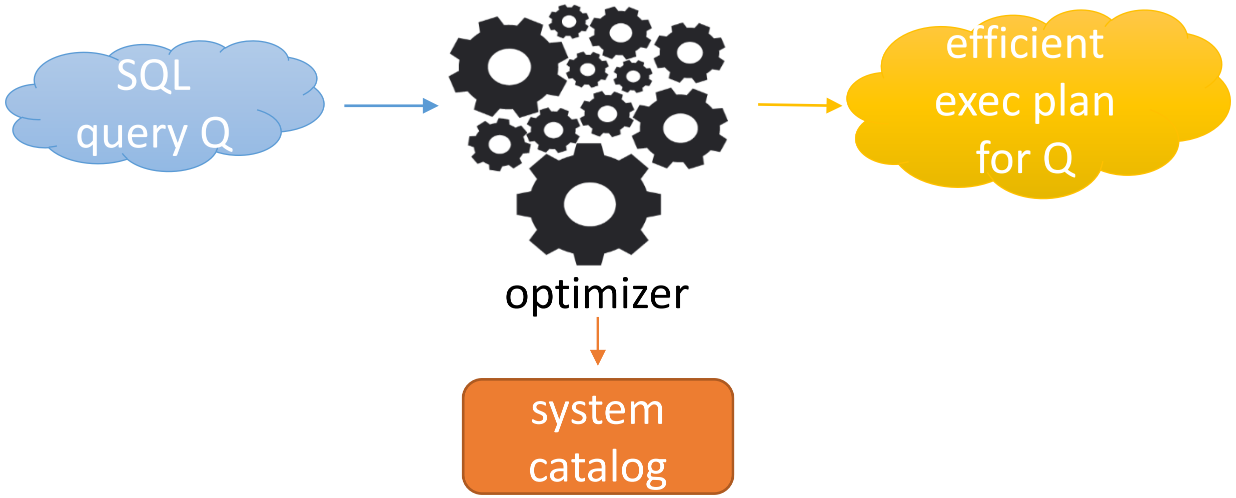
*Review lecture notes on *Relational Algebra* (*Databases* course)

* queries – composed of <u>relational operators</u>:

- intersection (R1 ∩ R2)
    - returns records that belong to both R1 and R2
- grouping and aggregate operators (algebra extensions)
- every operation returns a relation => operations can be composed
- an operator can have several implementation algorithms

* <u>optimizer</u>

- input: SQL query Q
- output: an efficient execution plan for evaluating Q



* algorithms for operators - based on 3 techniques:

- iteration:
  - examine iteratively:
    - all tuples in input relations

    or
    - data entries in indexes, provided they contain all the necessary fields (data entries are smaller than data records)
- indexing:
  - used when the query contains a selection condition or a join condition
  - examine only the tuples that meet the condition, using an index
- partitioning:
  - partition the tuples
  - decompose operation into collection of cheaper operations on partitions
  - partitioning techniques: sorting, hashing

* <u>access paths</u>

- *access path* = way of retrieving tuples from a relation
    - file scan

    or
    - an index *I* + a matching selection condition *C*
- condition *C* matches index *I* if *I* can be used to retrieve just the tuples satisfying *C*
- if relation *R* has an index *I* that matches selection condition *C*, then there are at least 2 access paths for *R* (file scan; index)

*Review lecture notes on *Indexes* (*Databases* course)

* <u>access paths</u> - example:

- relation *Students[SID, Name, City]*
- *I* - tree index on *Students* with search key <*Name*>
- query Q:

```
SELECT *
FROM Students
WHERE Name = 'Ionescu'
```

- condition *C*: *Name = 'Ionescu'*
- *C* matches *I*, i.e., index *I* can be used to retrieve only the *Students* tuples satisfying *C*
- the following condition also matches the index: *Name > 'Ionescu'*

* <u>access paths</u> - example:

- relation *Students[SID, Name, City]*
- *I* - hash index on *Students* with search key <*Name*>
- query Q:

  SELECT *

  FROM Students

  WHERE Name = 'Ionescu'
- condition *C*: *Name = 'Ionescu'*
- *C* matches *I*, i.e., index *I* can be used to retrieve only the *Students* tuples satisfying *C*
- condition *Name > 'Ionescu'* doesn't match *I* (since *I* is a hash index; it cannot be used to retrieve just the tuples satisfying *Name* > '*Ionescu*')

* <u>access paths</u>

- to sum up:
  - condition *C: attr op value*, *op* $\in$ *{<, <=, =, <>, >=, >}*
  - condition *C* matches index *I* if:
  - the search key of *I* is *attr* and:
    - *I* is a tree index or
    - *I* is a hash index and *op* is *=*

* access paths

- index *I*, selection condition *C*

- *I* - hash index
- condition *C* of the form:
  - $\bigwedge_{i=1}^{n} T_i$
  - term T_i: $attr = value$
- *I* matches *C* if *C* has one term for each attribute in the search key of *I*

Condition	Hash index with search key <a, b, c>
a = 10 AND b = 5 AND c = 2	Yes
a = 10 AND b = 5	No
b = 5	No
b = 5 AND c = 2	No

* access paths

- index *I*, selection condition *C*

- *I* - tree index
- condition *C* of the form:
    - $\bigwedge_{i=1}^{n} T_i$
    - term T_i: $attr\ op\ value$; *op* ϵ {<, <=, =, <>, >=, >}
- *I* matches *C* if *C* has one term for each attribute in a prefix of the search key of *I*

Condition	B+ tree index with search key <a, b, c>
a = 10 AND b = 5 AND c = 2	Yes
a = 10 AND b = 5	Yes
b = 5	No
b = 5 AND c = 2	No

* access paths

- selectivity of an access path
    - the number of retrieved pages when using the access path to obtain the desired tuples
    - both data pages and index pages are counted
- example:

SELECT *
FROM Students
WHERE Name = 'Ionescu'

  - access paths:
      - file scan – selectivity could be 1000
      - matching index I with search key <Name> – selectivity could be 3
- most selective access path
    - retrieves the fewest pages, i.e., data retrieval cost is minimized

* <u>general selection conditions</u>

- in general, a selection condition can contain one or several terms of the form:
  - *attr op constant*
  - *attr1 op attr2*,

  combined with ∧ and ∨

```
SELECT *
FROM Exams
WHERE SID = 7 AND EDate = '04-01-2021'
```

$$\sigma_{SID=7 \wedge EDate='04-01-2021'}(Exams)$$

* <u>general selection conditions</u>

- process a selection operation with a general selection condition C -> express C in CNF (conjunctive normal form)
- condition in CNF:
  - collection of conjuncts connected with the ∧ operator
  - a *conjunct* has one or more terms connected with the ∨ operator
  - *term*:
    - *attr op constant*
    - *attr1 op attr2*

- example:
  condition *(EDate < '4-1-2021' ∧ Grade = 10) ∨ CID = 5 ∨ SID = 3*
  is rewritten in CNF:
  (EDate < '4-1-2021' ∨ CID = 5 ∨ SID = 3) ∧ (Grade = 10 ∨ CID = 5 ∨ SID = 3)

* general selection conditions matching an index

- relation R[a, b, c, d, e], index I with search key <a, b, c>

Condition	B+ tree index	Hash index
a = 10 AND b = 5 AND c = 2	Yes	Yes
a = 10 AND b = 5	Yes	No
b = 5	No	No
b = 5 AND c = 2	No	No
d = 2	No	No
a = 20 AND b = 10 AND c = 5 AND d = 11	Partly	Partly

Condition – CNF selection condition

B+ tree index / Hash index – *B+ tree / hash index* I matches (*Yes*) / doesn't match (*No*) / matches a part of (*Partly*) the selection condition

- for the condition in the last row (a = 20 AND b = 10 AND c = 5 AND d = 11):
  - use index I to retrieve tuples satisfying *a = 20 AND b = 10 AND c = 5*, then apply *d = 11* to each retrieved tuple

* <u>general selection conditions</u> matching an index

- relation R[a, b, c, d]
- index I1 with search key <a, b>
- B+ tree index I2 with search key <c>

Condition	Indexes
c < 100 AND a = 3 AND b = 5	- use I1 or I2 to retrieve tuples - then check terms in the selection condition that do not match the index for each retrieved tuple - e.g., use the B+ tree index to retrieve tuples where c < 100; then apply *a = 3 AND b = 5* to each retrieved tuple

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

* <u>joins</u>

```
SELECT *
FROM Exams E, Students S
WHERE E.SID = S.SID
```

- algebra: $E \otimes_{E.SID=S.SID} S$
    - to be carefully optimized
    - size of $E \times S$ is large, so computing $E \times S$ followed by selection is inefficient
- E
    - M pages
    - p_E records / page
- S
    - N pages
    - p_S records / page
- evaluation: number of I/O operations

* <u>joins</u> – implementation techniques
    - iteration
        - Simple/Page-Oriented Nested Loops Join
        - Block Nested Loops Join
    - indexing
        - Index Nested Loops Join
    - partitioning
        - Sort-Merge Join
        - Hash Join

- equality join, one join column
    - join condition: $E_i = S_j$

Simple Nested Loops Join

```
foreach tuple e ∈ E do
    foreach tuple s ∈ S do
        if e_i == s_j then add <e, s> to the result
```

- for each record in the outer relation E, scan the entire inner relation S
- cost
  - **M + p_E* M * N** = 1000 + 100*1000*500 I/Os = 1000 + (5 * 10^7) I/Os
    - M I/Os – cost of scanning E
    - N I/Os – cost of scanning S
    - S is scanned p_E* M times (there are p_E* M records in the outer relation E)

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

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

Page-Oriented Nested Loops Join

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if e_i == s_j then add <e, s> to the result
```

- for each page in E read each page in S
- pairs of records <e, s> that meet the join condition are added to the result (where record *e* is on page *pe*, and record *s* – on page *ps*)
- refinement of Simple Nested Loops Join

Page-Oriented Nested Loops Join

```
foreach page pe ∈ E do
    foreach page ps ∈ S do
        if e_i == s_j then add <e, s> to the result
```

- <u>cost</u>
  - **M + M*N** = 1000 + 1000*500 I/Os = 501.000 I/Os
    - M I/Os – cost of scanning E; N I/Os – cost of scanning S
    - S is scanned M times
    - significantly lower than the cost of Simple Nested Loops Join (improvement - factor of p_E)
  - if the smaller table (*S*) is chosen as outer table:

  => cost = 500 + 500 * 1000 I/Os = 500.500 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 *

<u>Block Nested Loops Join</u>

- previously presented join algorithms do not use buffer pages effectively
- join relations R1 and R2; R1 – the smaller relation
- assumption – the smaller relation fits in main memory
- <u>improvement</u>:
    - store smaller relation R1 in memory
    - keep at least 2 extra buffer pages B1 and B2
    - use B1 to read the larger relation R2 (one page at a time)
    - use B2 as the output buffer (i.e., for tuples in the result of the join)
    - for each tuple in R2, search R1 for matching tuples

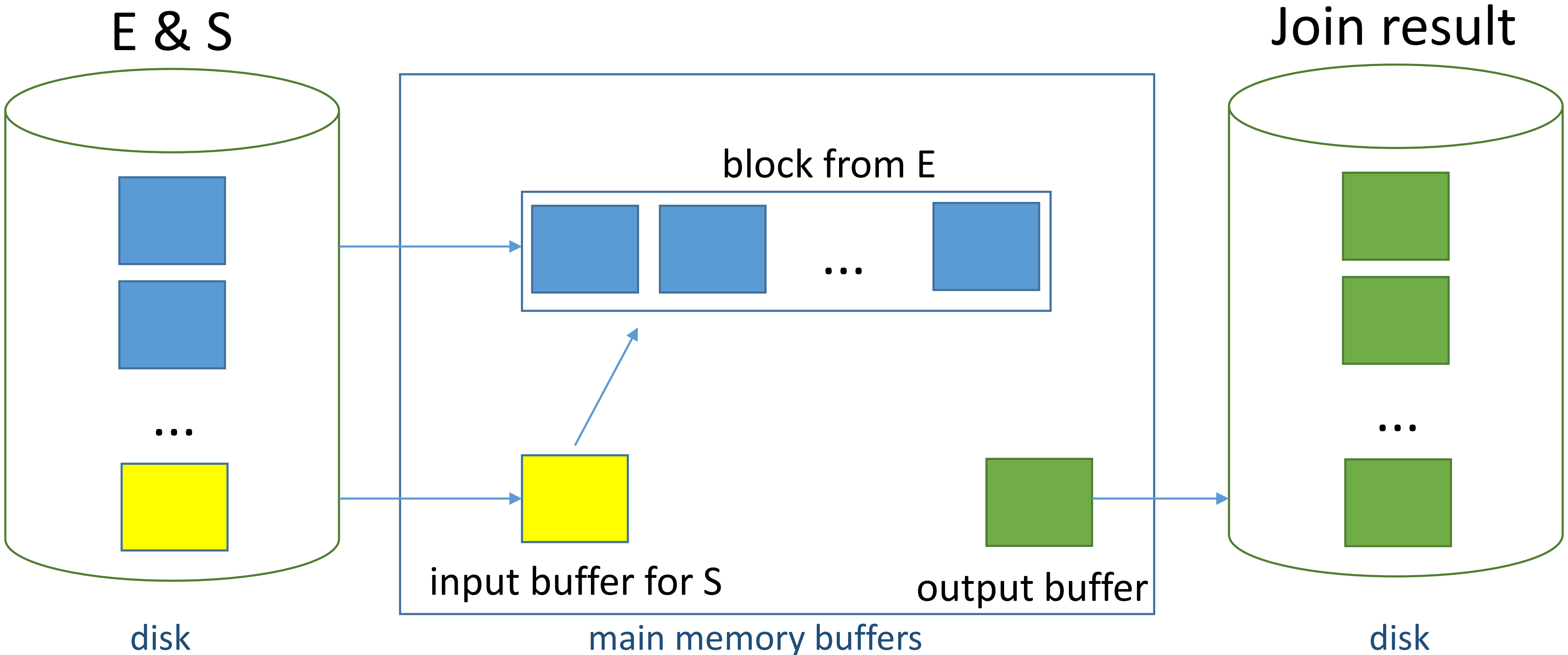
  => optimal cost: *number of pages in R1 + number of pages in R2*, since R1 is scanned only once, R2 is also scanned only once

Block Nested Loops Join

- refinement
  - don’t store the smaller relation in main memory as is, build an in-memory hash table for it instead
  - the I/O cost remains unchanged, but the CPU cost is usually much lower (since for each tuple in the larger relation, the smaller relation is examined to find matching tuples)

Block Nested Loops Join

- if there isn't enough main memory to hold one of the input relations:
  - use one buffer page to scan the inner table (e.g., S)
  - use one page for the result
  - use all remaining pages to read a *block* from the outer table (e.g., E)
    - block – set of pages from E that fit in main memory



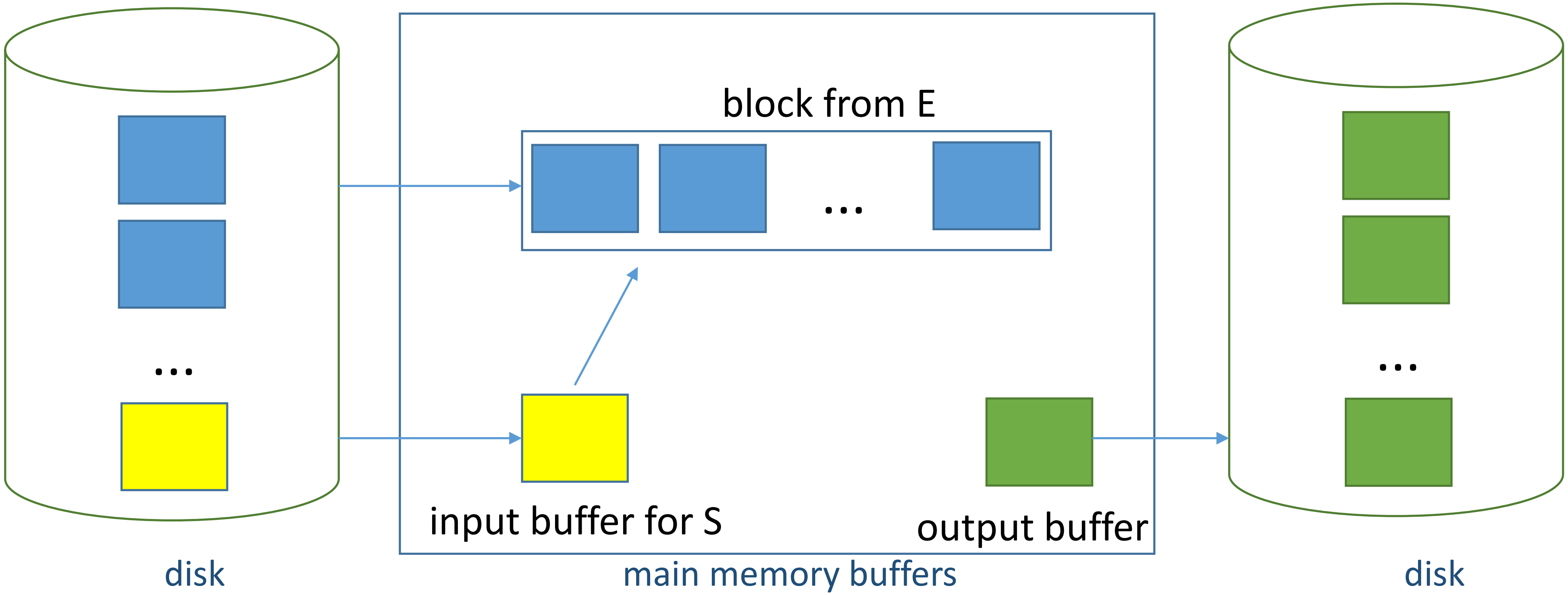
Block Nested Loops Join

```
foreach block be ∈ E do
    foreach page ps ∈ S do
      {
          for all pairs of tuples <e, s> that meet the join
            condition, where e ∈ be and s ∈ ps,
                 add <e, s> to the result
      }
```

- inner relation S is scanned once for each block in outer relation E
- outer relation E is scanned once

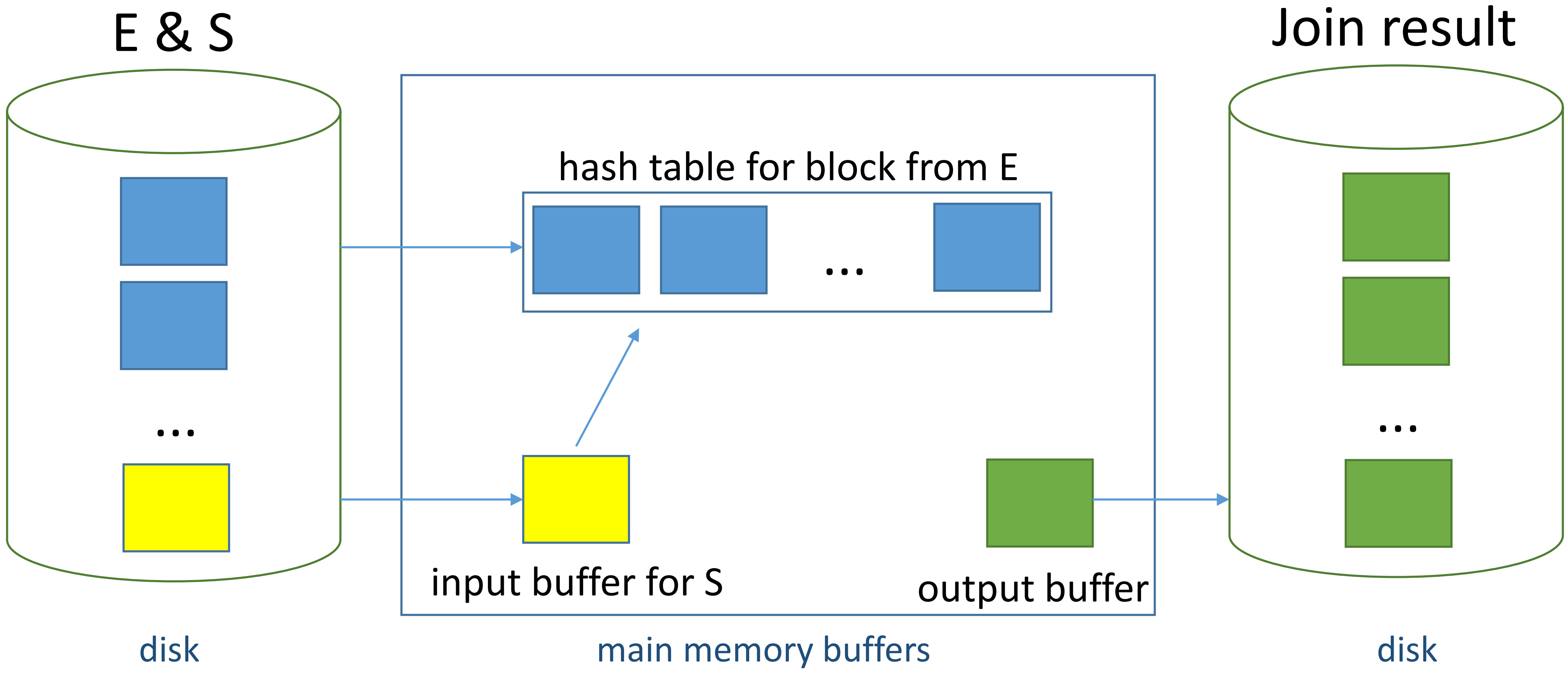
E & S

Join result



Block Nested Loops Join

- refinement to efficiently find matching tuples
  - build main-memory hash table for the block of E
  - trade-off: reduce size of E block



Block Nested Loops Join

- cost
  - scan of outer table + number of blocks in outer table * scan of inner table
  - number of outer blocks = $\left\lceil \frac{\text{number of pages in outer table}}{\text{size of block}} \right\rceil$
  - outer table: Exams (E), a block can hold 100 pages
    - scan cost for E: 1000 I/Os
    - number of blocks: $\left\lceil \frac{1000}{100} \right\rceil$ = 10
    - foreach block in E, scan Students (S): 10*500 I/Os

  => total cost = 1000 + 10 * 500 = **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 *

Block Nested Loops Join

- cost
  - scan of outer table + number of blocks in outer table * scan of inner table
  - number of outer blocks = $\left\lceil \frac{\text{number of pages in outer table}}{\text{size of block}} \right\rceil$
  - outer table: Exams (E)
    - suppose the buffer has 90 pages available for E, i.e., block of 90 pages

    => number of blocks: $\left\lceil \frac{1000}{90} \right\rceil$ = 12

    => S is scanned 12 times
    - scan cost for E: 1000 I/Os
    - foreach block in E, scan Students (S): 12*500 I/Os

  => total cost = 1000 + 12 * 500 = **7000 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 *

Block Nested Loops Join

- cost
  - scan of outer table + number of blocks in outer table * scan of inner table
  - number of outer blocks = $\left\lceil \frac{\text{number of pages in outer table}}{\text{size of block}} \right\rceil$
  - outer table: Students (S), block of 100 pages
    - scan cost for S: 500 I/Os
    - number of blocks: $\left\lceil \frac{500}{100} \right\rceil$ = 5
    - for each block in S, scan E: 5 * 1000 I/Os

  => total cost = 500 + 5 * 1000 = **5500 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 *

Index Nested Loops Join

```
foreach tuple e in E do
    foreach tuple s in S where e_i == s_j
        add <e, s> to the result
```

- if there is an index on the join column of S, S can be considered as inner table and the index can be used
- cost
  - $M + ((M*p_E) *$ cost of finding corresponding records in S)

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

Index Nested Loops Join

- for a record e in E:
  - the cost of examining the index on S is:
    - approx. 1.2 for a hash index (typical cost for hash indexes)
    - typically 2-4 for a B+-tree index

  - the cost of reading corresponding records in S:
    - for a clustered index:
      - plus one I/O for each outer tuple in E (typically)
    - for an unclustered index:
      - up to one I/O for each corresponding record in S
        (worst case – there are *n* matching records in S located on *n* different pages!)

Index Nested Loops Join

- hash index on SID in Students (Students – inner table)
- scan Exams:
    - cost = 1000 I/Os, with a total of 100*1000 records
- for each record in Exams:
    - (on average) 1.2 I/Os to obtain the page in the hash index (i.e., the page containing the rid of the matching Students tuple)
      and
    - 1 I/O to retrieve the page in Students that contains the matching tuple (exactly one! – since SID is a key in Students, i.e., there is one matching Students tuple for an exam)

=> cost to retrieve matching Students tuples: 1000 * 100 * (1.2 + 1) = 220.000

- total cost: 1000 + 220.000 = 221.000 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 *

* <u>sorting</u>

- can be explicitly required (SELECT ... ORDER BY list), used to eliminate duplicates (SELECT DISTINCT), used by operators like:
  - join
  - union
  - intersection
  - set-difference
  - grouping

* <u>sorting</u>

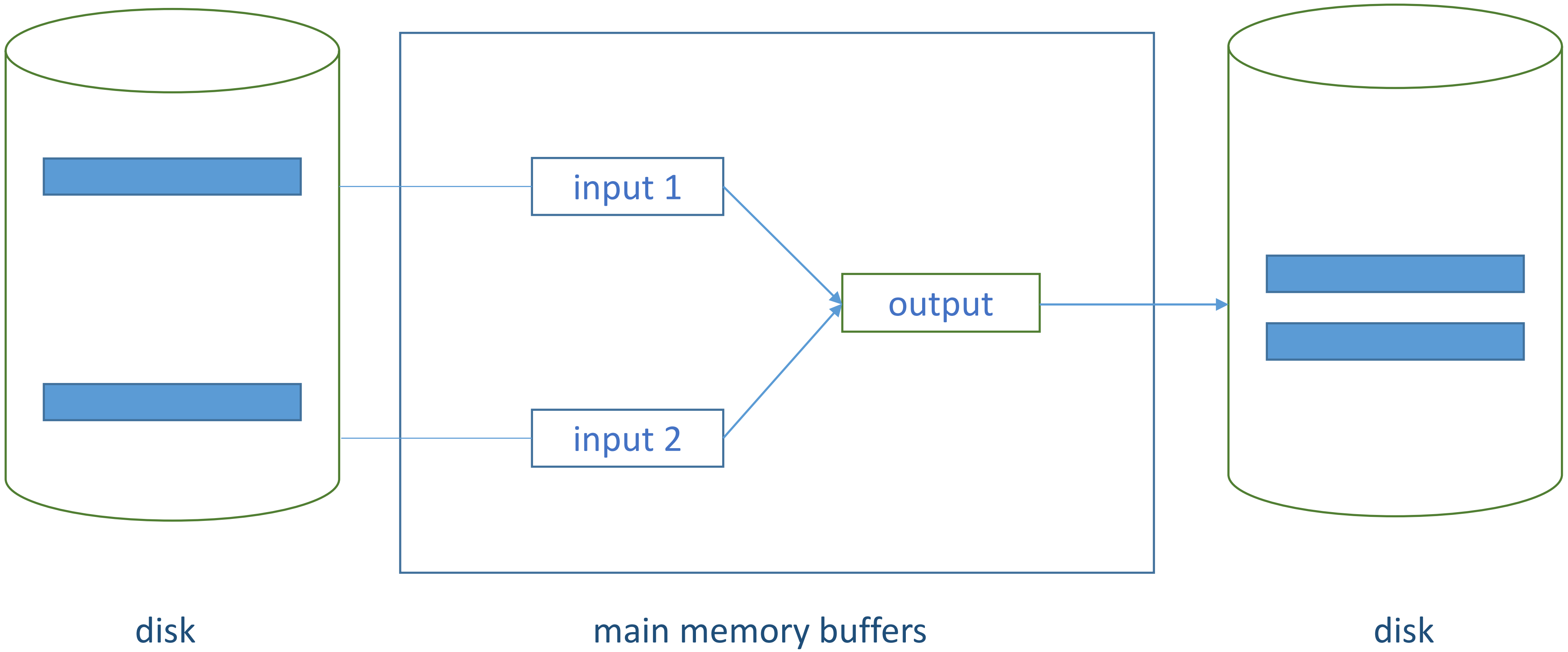
e.g., the user wants to sort the collection of Courses records by name

- 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
    - 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
    - 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 1^{st} 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 2^{nd} page from Courses, sort the 80 records on it by course name, write out sorted page to disk;

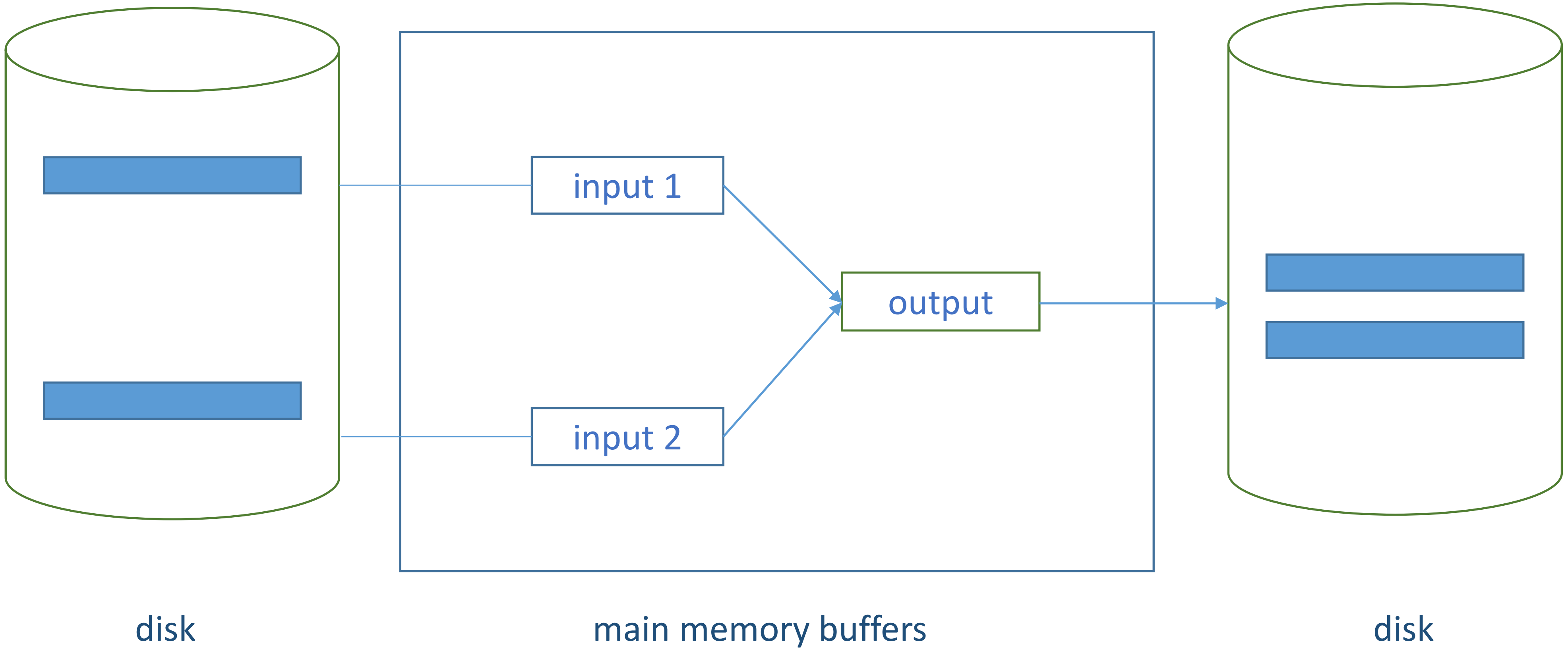
    …

    - read in the 100^{th} 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

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



Simple Two-Way Merge Sort

- e.g., pass 1 (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

- <u>pass 0</u>
  - 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: [3, 4] [2, 6] [4, 9] [7, 8] [5, 6] [1, 3] [2]

- pass 1
  - read in runs [5, 6] and [1, 3] …

    => run [1, 3 | 5, 6]

  - read in run [2] (the last run from pass 0) …

    => run [2]

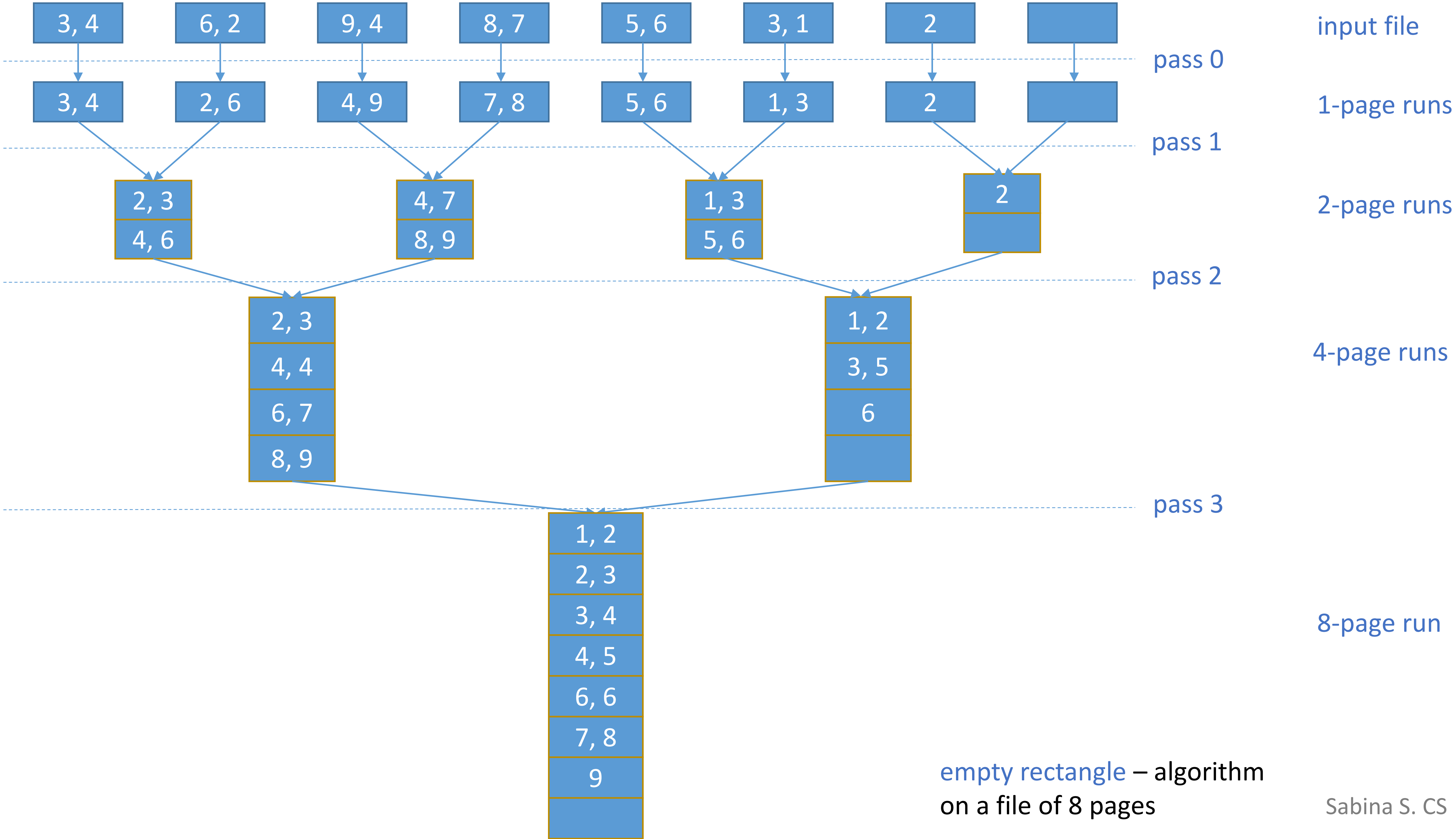
  => 4 sorted runs that are 2 pages long (except for the last run):

  [2, 3 | 4, 6] [4, 7 | 8, 9] [1, 3 | 5, 6] [2]

Simple Two-Way Merge Sort

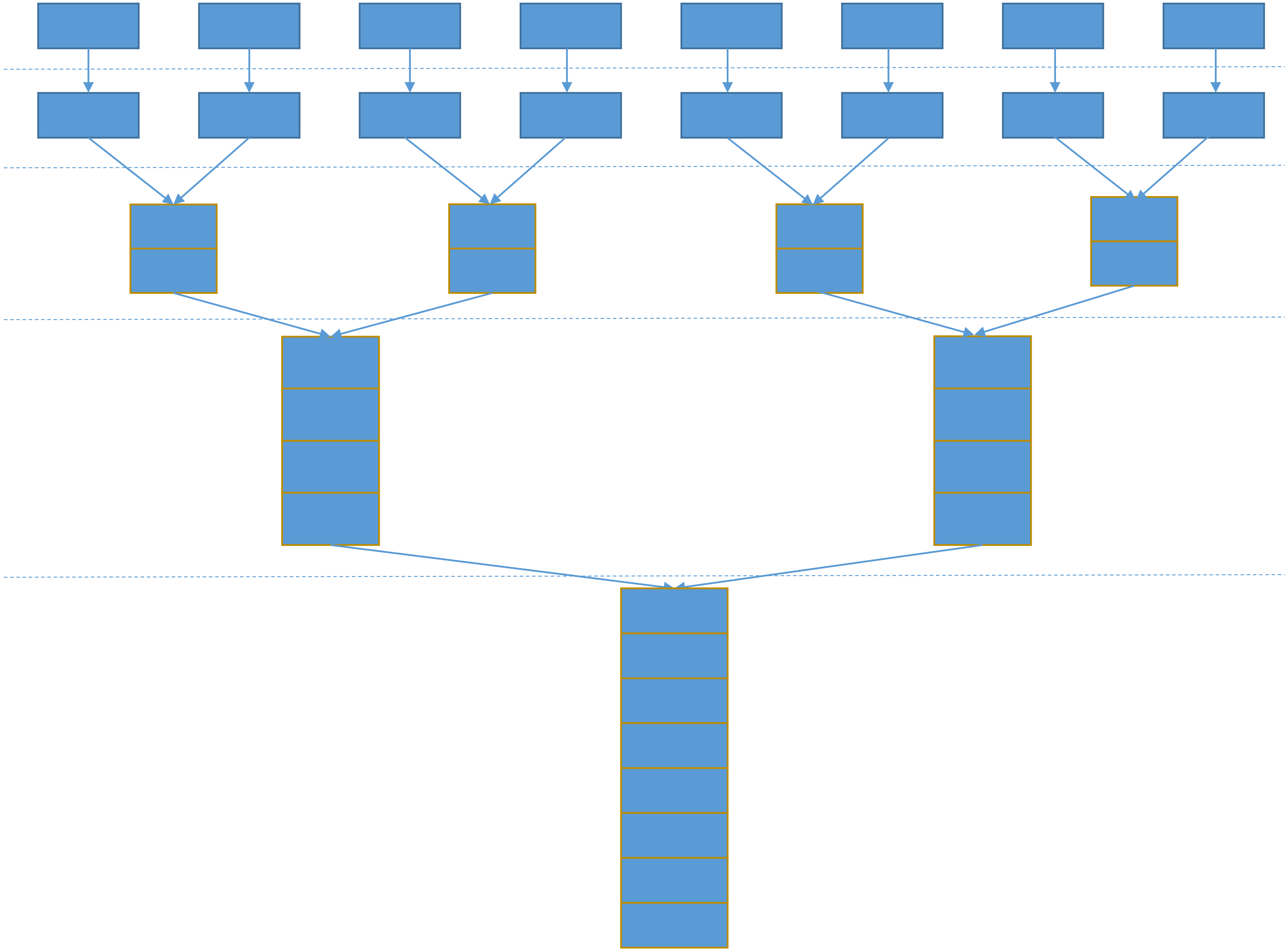
- pass 2
  - read in & merge pairs of runs from pass 1
  - produce runs that are twice as long

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



empty rectangle – algorithm
on a file of 8 pages

Sabina S. CS



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)

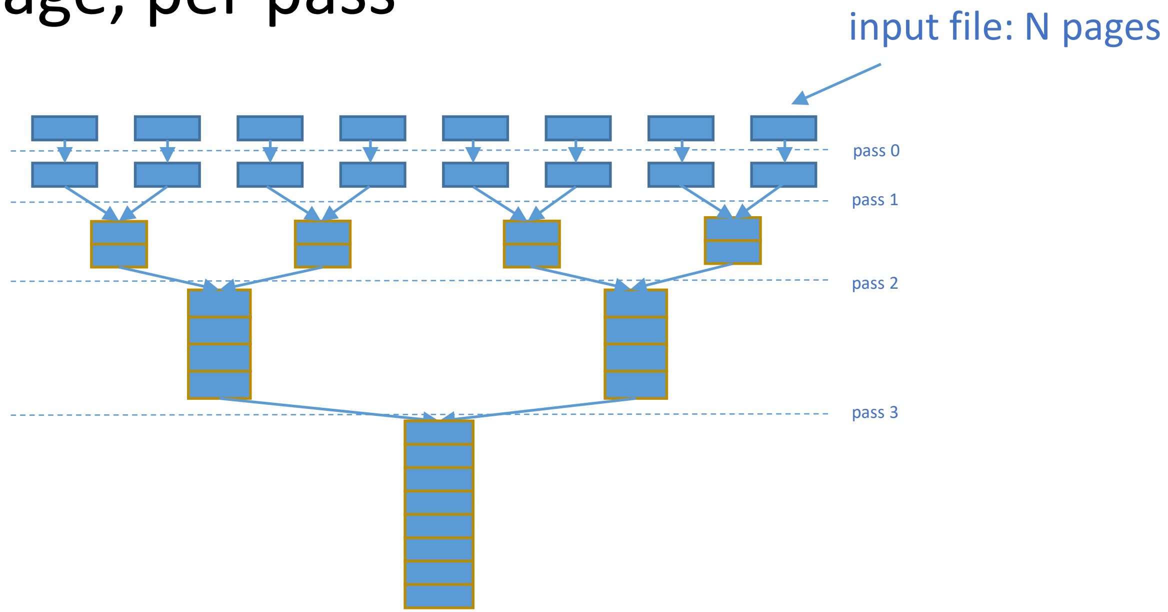
pass 3

=> 2^{k-3} sorted runs (8-pages)

i.e.,
pass k
=> one sorted run (2^k pages)

Simple Two-Way Merge Sort

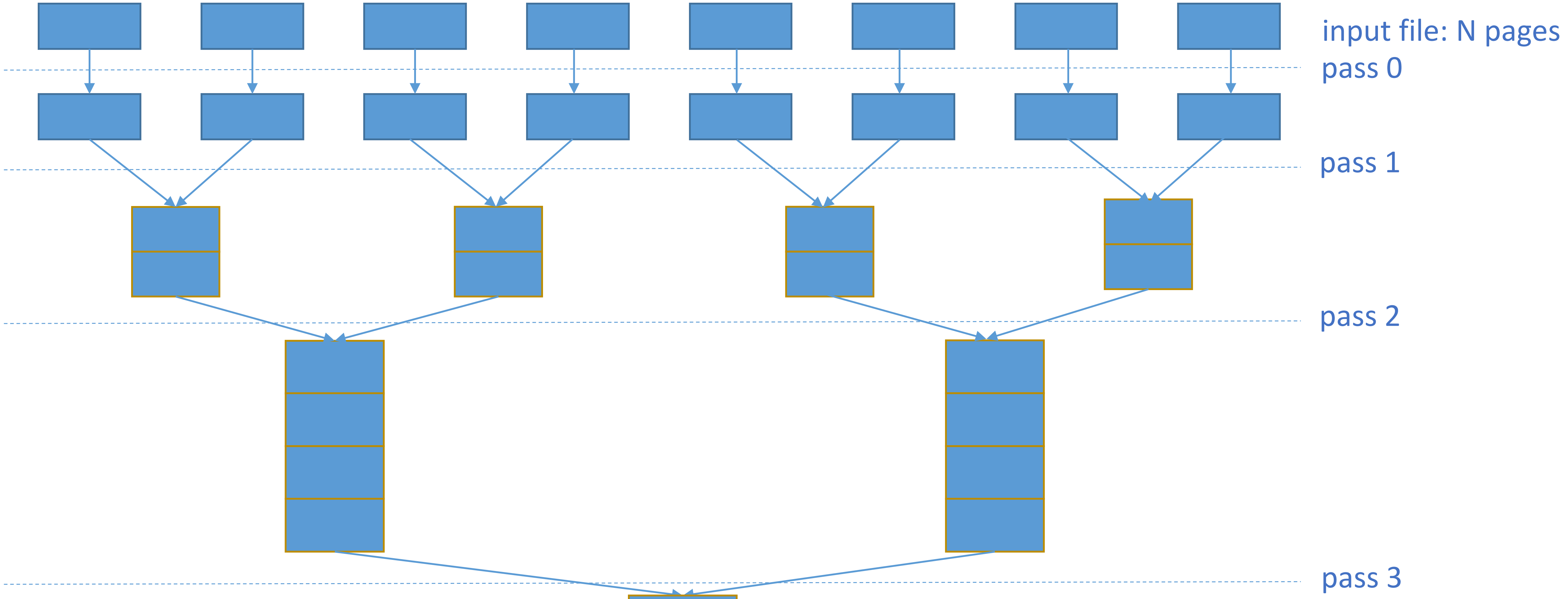
- in each pass, each page in the input file is: read in, processed, and written out; there are 2 I/O operations per page, per pass



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

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

- total <u>cost</u>:
  - 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
    - $2 * 8 * (\lceil \log_2 8 \rceil + 1) = 2 * 8 * 4 = 64$ I/Os
- N = 7 pages, 4 passes
  => 2 * 7 * 4 = 56 I/Os
    - $2 * 7 * (\lceil \log_2 7 \rceil + 1) = 56$ 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

- input file to be sorted: N pages
- B buffer pages are available
- pass 0:
  - 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:

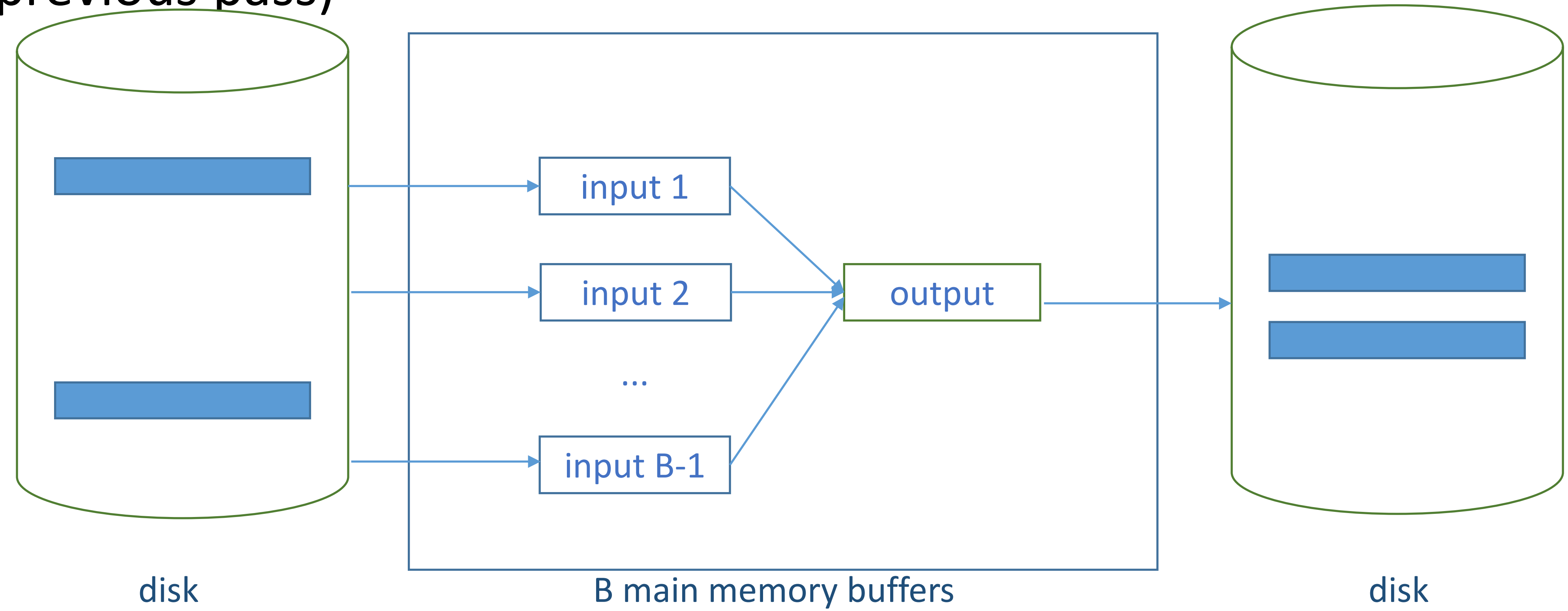
5, 6	3, 1	2

  - 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
  - in 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

External Merge Sort – 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

References

- [Ra02] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems (3rd Edition), McGraw-Hill, 2002
- [Da03] DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2003
- [Ga09] GARCIA-MOLINA, H., ULLMAN, J., WIDOM, J., Database Systems: The Complete Book (2nd Edition), Pearson Education, 2009
- [Ra02S] RAMAKRISHNAN, R., GEHRKE, J., Database Management Systems, Slides for the 3rd Edition, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
- [Si19] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts (7th Edition), McGraw-Hill, 2019
- [Si19S] SILBERSCHATZ, A., KORTH, H., SUDARSHAN, S., Database System Concepts, Slides for the 7th Edition, 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