Database Design and Implementation

CS 645

Query evaluation and optimization

Access methods

- Routines that manage disk-based data structures

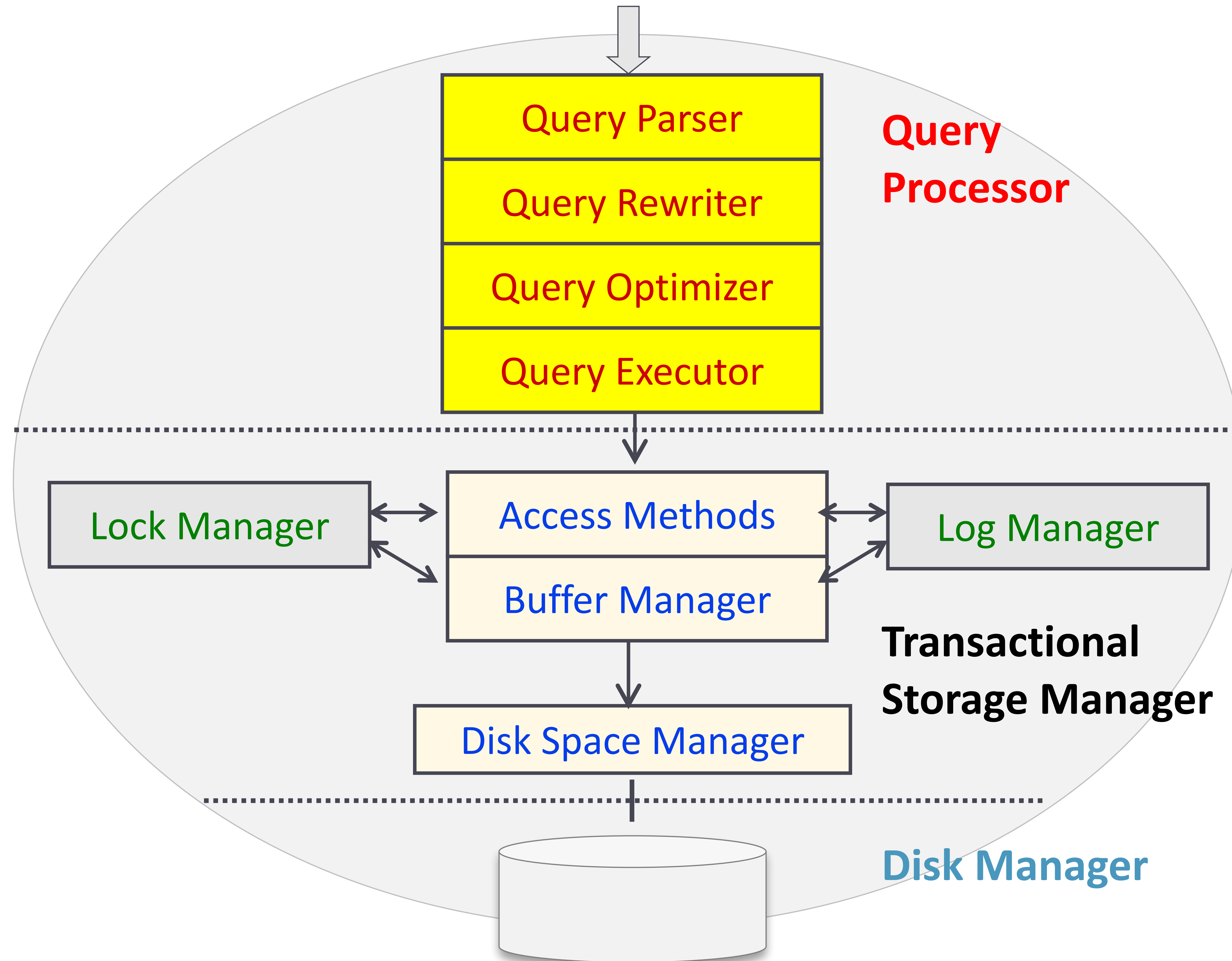
- File of records:
  - Abstraction of external storage for query processing
  - (1) Sequential scan; (2) Locate a record using record id (rid)

    E.g., retrieve all sailor records, or a record w. (page 4, slot 2)

- Indexes:
  - Auxiliary data structures
  - Associative access: given a value in the index search key, find the (record ids of) records with this value.

    E.g., find all sailors with rating > 5.

DBMS architecture



Relational operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Join* (⋈) Allows us to combine two relations.
  - *Projection* (π) Deletes unwanted columns from relation.

  - *Union* (U) Tuples in either reln. 1 or reln. 2.
  - *Intersection* (∩) Tuples in both reln. 1 and reln. 2.
  - *Set-difference* (−) Tuples in reln. 1, but not in reln. 2.

  - *GROUP BY* and *Aggregation* (SUM, MIN, etc.)

Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

Schema for examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: date, *rname*: string)

- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.
- Reserves:
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

- Cost metric: # I/Os

Using an index for selections

```
SELECT *
FROM   Sailors S
WHERE  S.rating > 8
```

- Cost of selection includes:
  1) top down search in the index
  2) scan the relevant leaf nodes
  3) retrieve records from file (could be large w/o clustering).

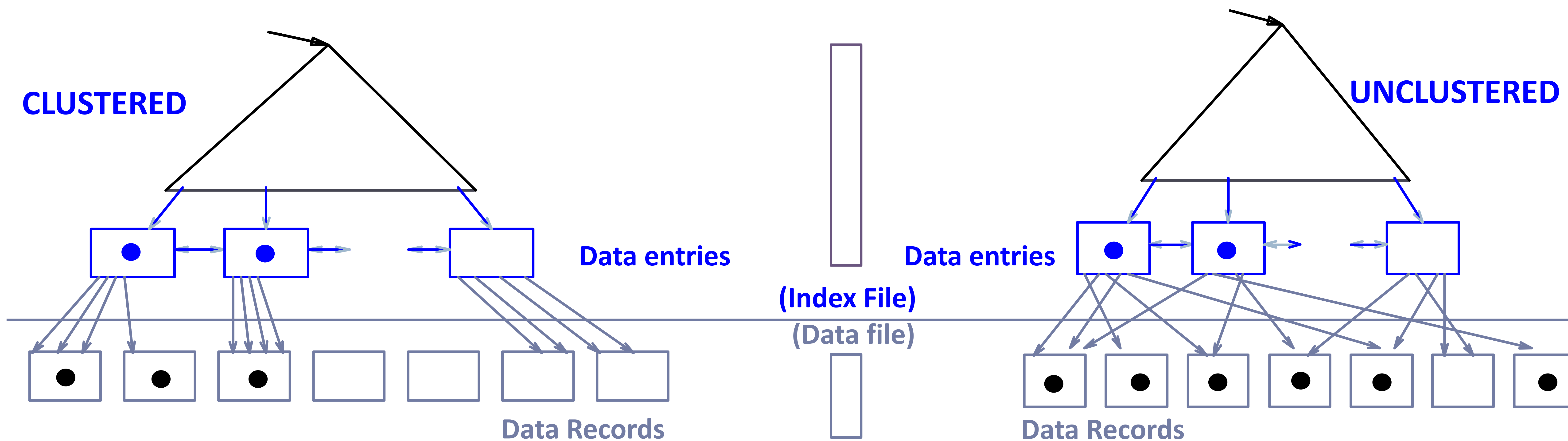
- Step 1— top down search: ≤ 3-4 I/Os, depending on buffer management

Cost factors of steps 2 and 3

```
SELECT  *
FROM    Sailors S
WHERE   S.rating > 8
```

- Cost of selection includes:
  1) top down search in the index
  2) scan the relevant leaf nodes
  3) retrieve records from file (could be large w/o clustering).
- Cost factor: *number of qualifying tuples*
  - *rating* > 8: if 20% of tuples qualify, 500/5=100 pages, 80*100=8,000 tuples.
  - **Scanning leaf nodes**: if a data entry is 1/5 of a tuple, need 20 leaf nodes, so 20 I/Os.

Cost factors of selection



- Cost factor: *clustering*
  - *rating > 8*: 20% of tuples qualify, 100 pages, 8,000 tuples.
  - **Retrieving records from file ≈**
    - Clustered index: 100 I/Os.
    - Unclustered index: worst case 1 I/O per tuple; 8,000 I/Os here!

General selections

- Boolean combination of predicates using AND and OR.
  - Conjunctive Normal Form (CNF), e.g.,

    pred1 AND (pred3 OR pred4)

    (pred1 OR pred2) AND (pred3 OR pred4)

- *File scan* always works for general selections.

- *Index scan* works when it matches a predicate that is a conjunct of CNF.
  - E.g., an index matching *pred1* can be used for

    pred1 AND (pred3 OR pred4)

Conjunctive predicates only

- CNF without OR: e.g., *pred 1 AND pred 2 AND pred 3*
  - Find the ***most selective access path***, retrieve tuples using it
    - File scan or index scan that gives the smallest I/O cost.
    - Apply remaining terms that don't match index *on the fly*.
    - Other terms do not affect I/O cost.

day<8/9/94 AND bid=5 AND sid=3

- Hash index on ⟨***bid, sid***⟩: check ***day<8/9/94*** on the fly.
- B+ tree index on ***day***: apply ***bid=5*** and ***sid=3*** on the fly.

Improvement: intersection of rids

- 2+ matching indexes (Alternatives 2 or 3):
  1. Get sets of rids of data records using each index.
  2. *Intersect* these sets of rids.
  3. Retrieve the records and apply any remaining terms.

day<8/9/94 AND bid=5 AND sid=3

B+ tree index on *day*, a hash index on *sid*, both using Alt 2:

1. retrieve rids of records satisfying *day<8/9/94* using first index, rids of records satisfying *sid=3* using second index,
2. intersect these rids,
3. retrieve records, check *bid=5*.

Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

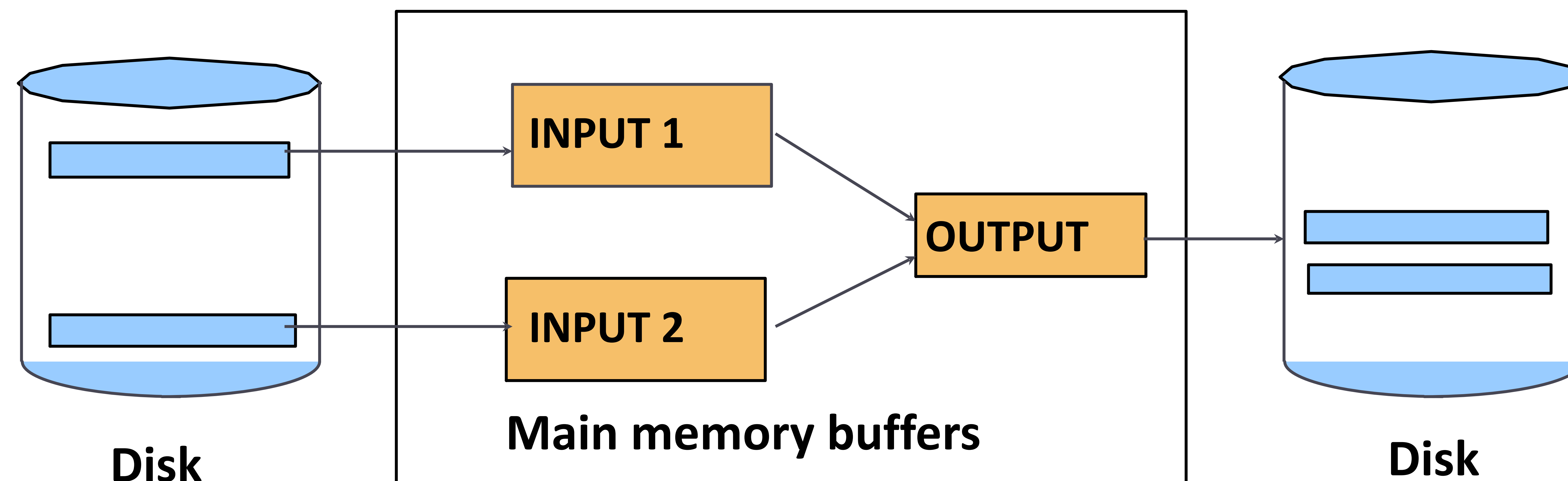
Why sort?

- Important utility in DBMS:
  - *Sort-merge* join algorithm involves sorting.
  - *Eliminate duplicates* in a collection of records (e.g., SELECT DISTINCT)
  - Request data in *sorted order* (e.g., ORDER BY)
    - e.g., find students in decreasing order of *gpa*
  - Sorting is first step in *bulk loading* B+ tree index.

- **Problem**: sort 1GB of data with 1MB of RAM.
  - Limited Memory. Key is to minimize # I/Os!

Two-way sort: Requires 3 buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.: Merge two sorted subfiles
  - three buffer pages used.



Two-way external merge sort

Divide and conquer:

- sort subfiles (runs)
- and merge

A file of N pages:

Pass 1: N sorted runs of 1 page each

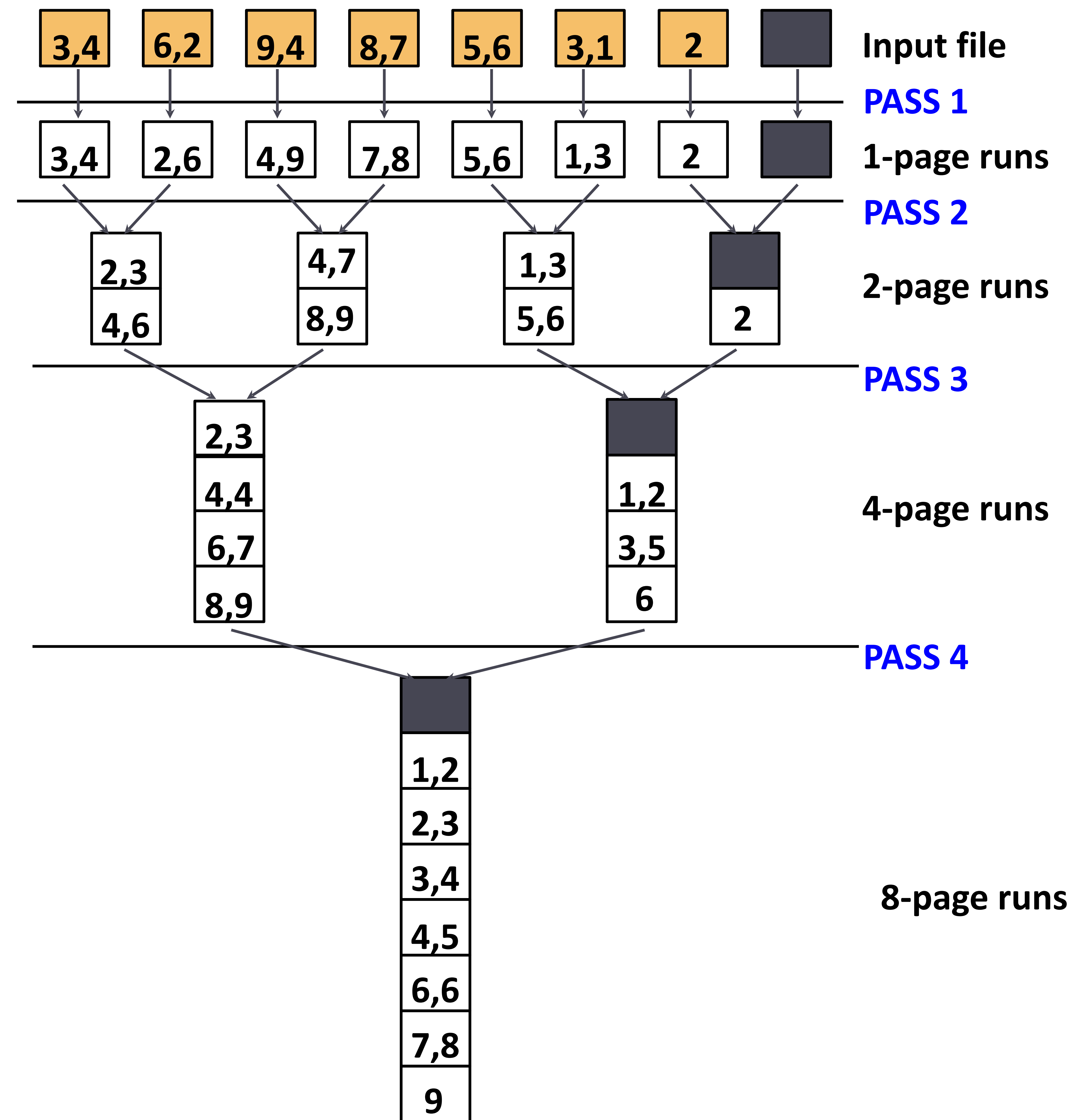
Pass 2: N/2 sorted runs of 2 pages each

Pass 3: N/4 sorted runs of 4 pages each

...

Pass P+1: 1 sorted run of 2^P pages

$2^P \geq N$ → $P \geq \log_2 N$



Two-way external merge sort

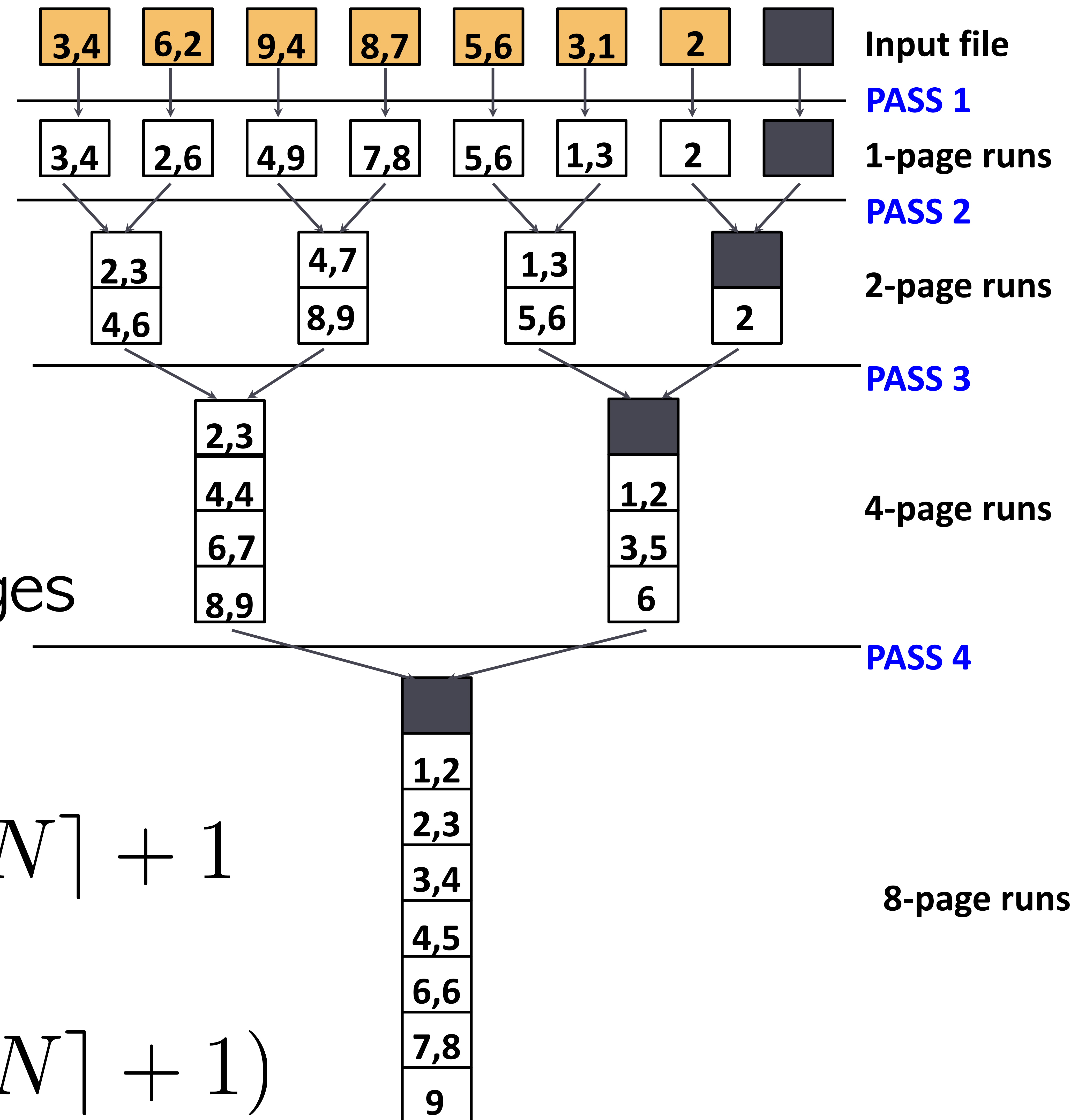
Divide and conquer:

- sort subfiles (runs)
- and merge

- Each pass, read + write N pages in file → 2N.

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

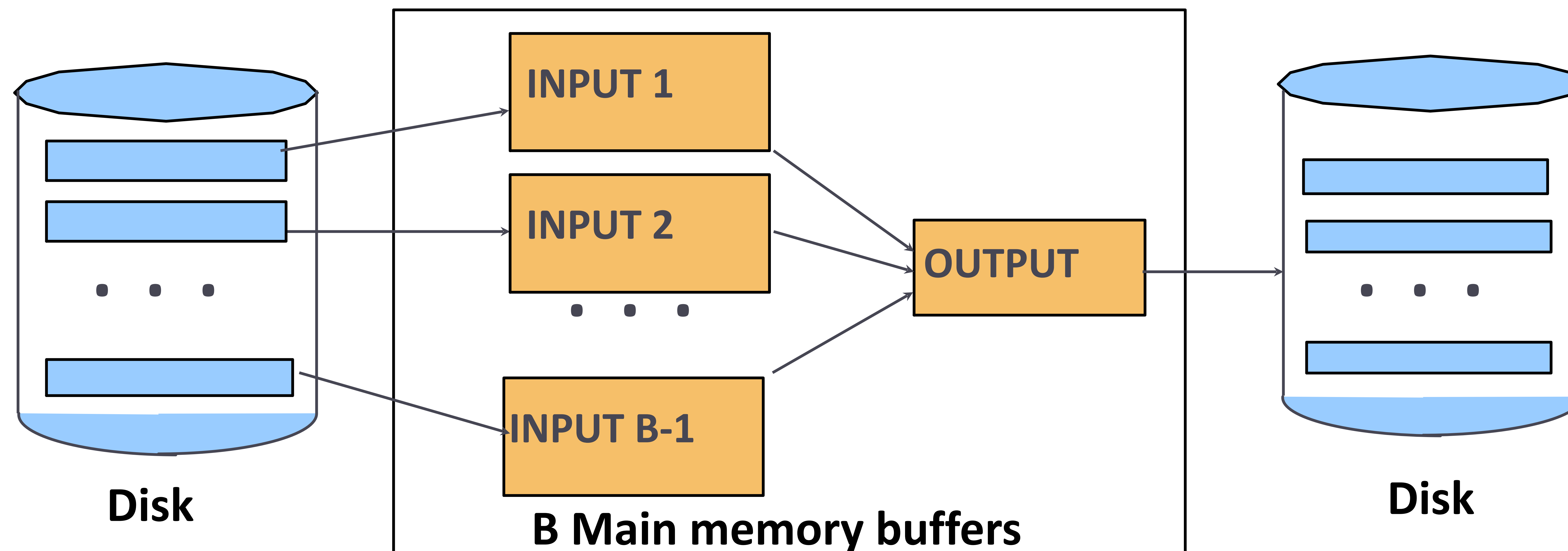
- So total cost is: $2N(\lceil \log_2 N \rceil + 1)$



General external merge sort

Given B (>3) buffer pages. How can we utilize them?

- **Pass 1**: Use *B* buffer pages. Produce ⌈N/B⌉ sorted runs of *B* pages each.
- **Pass 2, 3…, etc.**: Merge *B-1* runs.



Cost of external merge sort

- E.g., with 5 (B) buffer pages, sort 108 (N) page file:

Pass 1	$\lceil 108/5 \rceil = 22$ sorted runs of 5 pages each (last run is only 3 pages)	$\lceil N/B \rceil$ sorted runs of B pages each
Pass 2	$\lceil 22/4 \rceil = 6$ sorted runs of 20 pages each (last run is only 8 pages)	$\lceil N/B \rceil /(B-1)$ sorted runs of $B(B-1)$ pages each
Pass 3	2 sorted runs, 80 pages and 28 pages	$\lceil N/B \rceil /(B-1)^2$ sorted runs of $B(B-1)^2$ pages
Pass 4	Sorted file of 108 pages	$\lceil N/B \rceil /(B-1)^3$ sorted runs of $B(B-1)^3$ ($\geq N$) pages

- Number of passes $= 1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

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

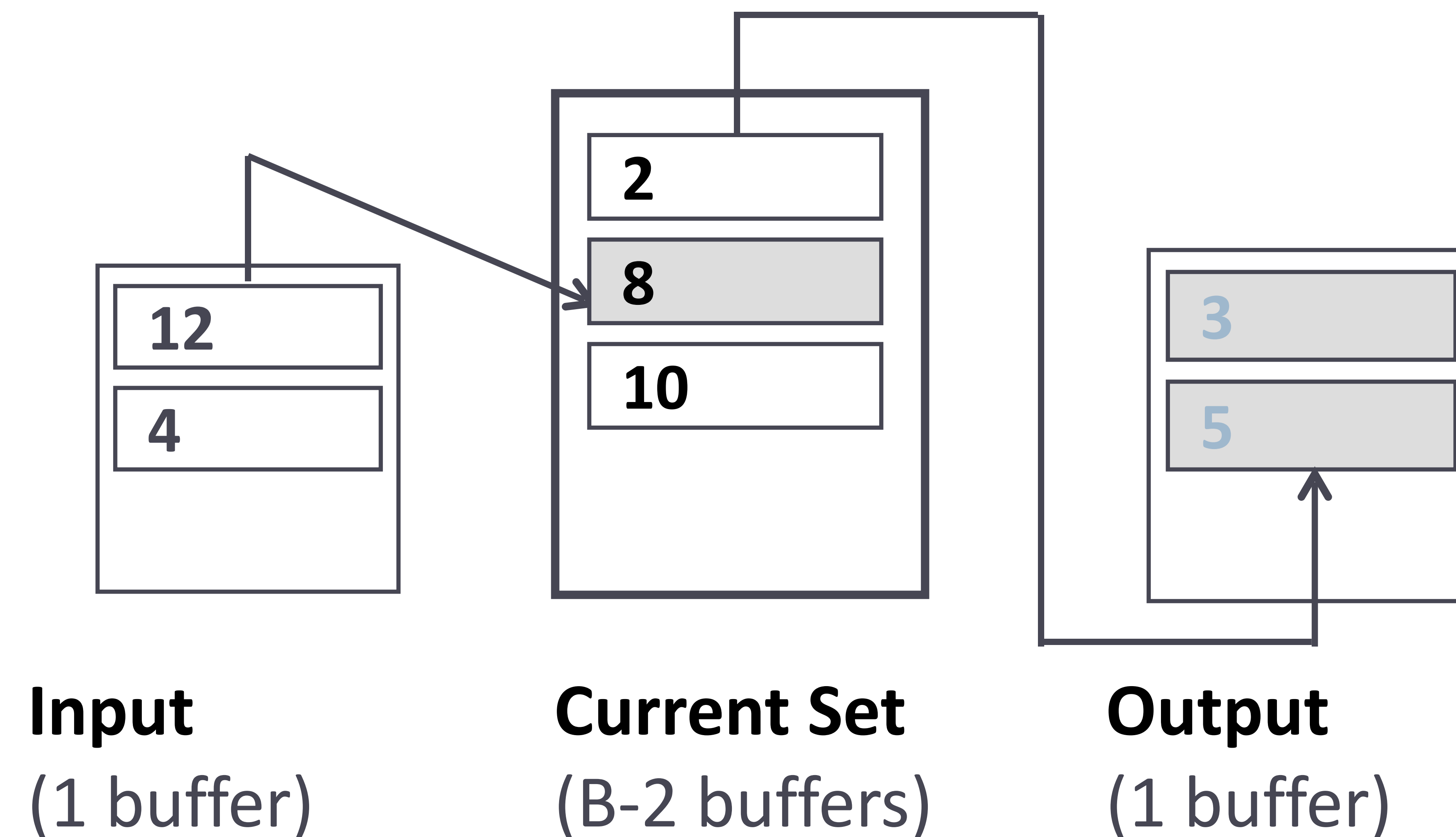
Number of passes of external sort

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

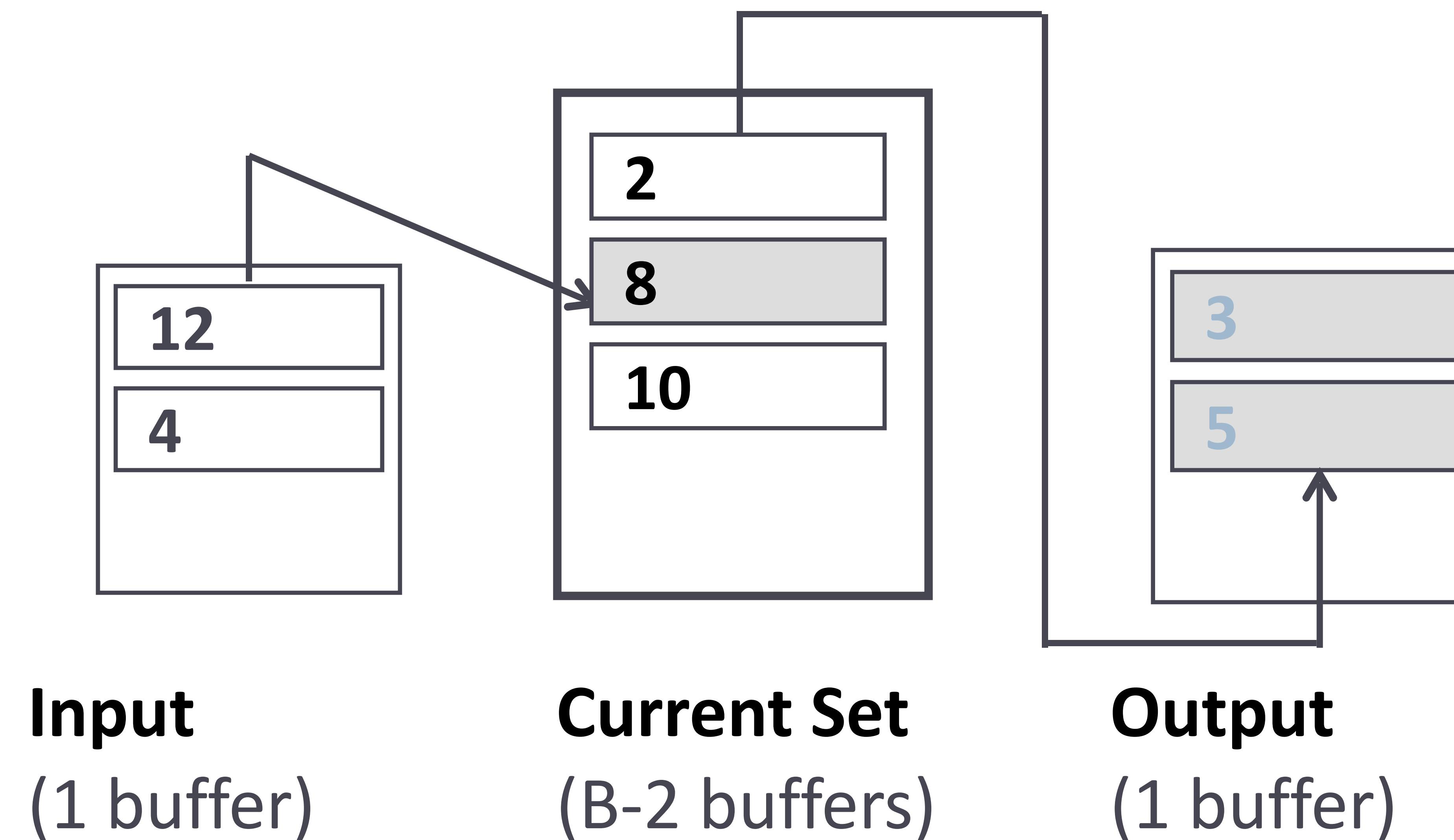
Replacement sort

- *Replacement sort*: Produce initial sorted runs as long as possible. Used in Pass 1 of sorting.

- Organize B available buffers:
  - 1 buffer for *input*
  - B-2 buffers for *current set*
  - 1 buffer for *output*

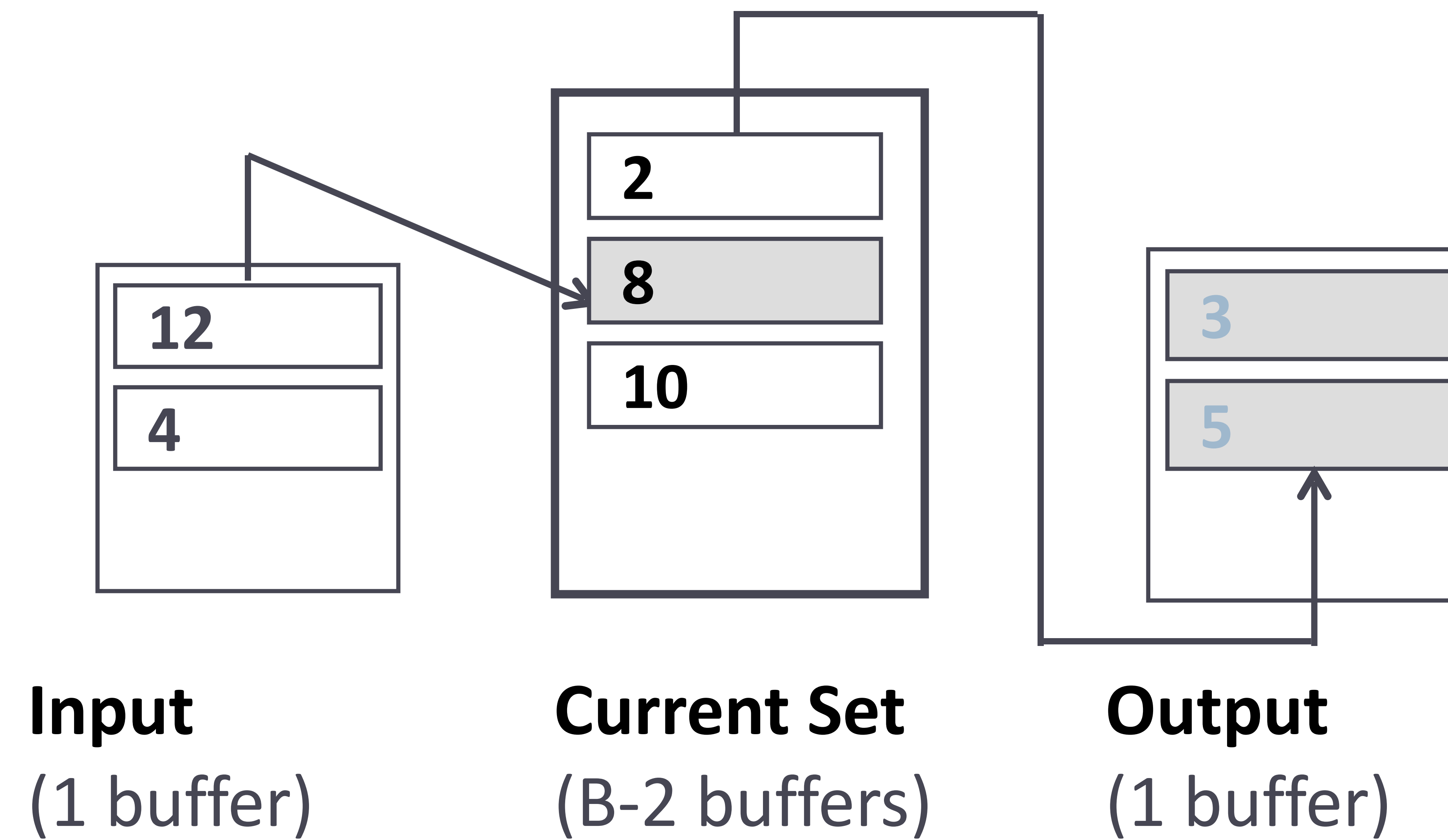


Replacement sort



- Pick tuple r in the current set (CS) s.t. r is the smallest value in CS that is $\geq$ largest value in output, e.g. 8, to extend the current run.
- Write output buffer out if full, extending the current run.
- Fill the space in current set by adding tuples from input.
- Current run terminates if every tuple in the current set is $<$ the largest tuple in output.

Replacement sort



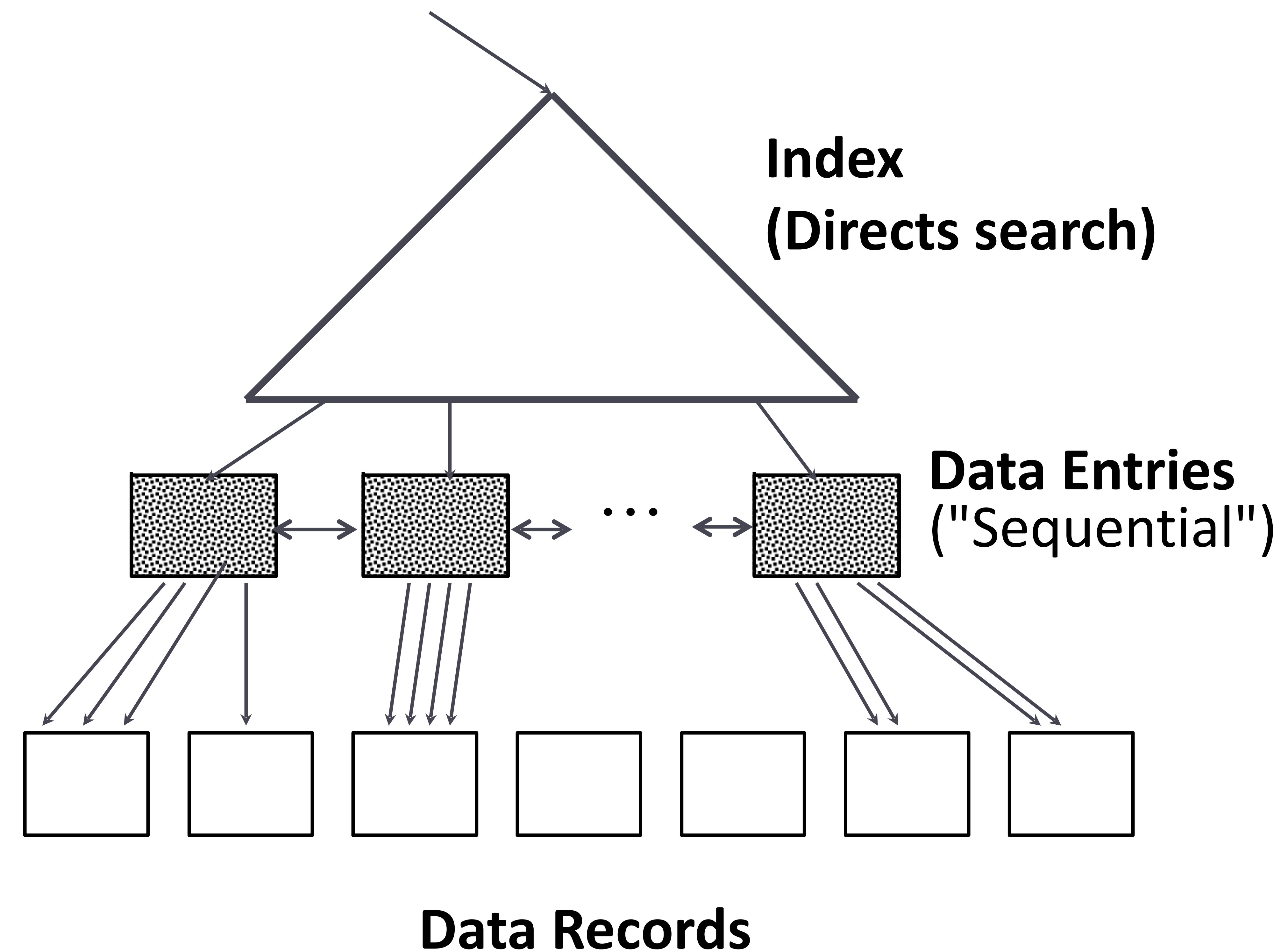
- When used in Pass 1 for sorting, write out sorted runs of size 2B on average.
- Affects calculation of the number of passes accordingly.

Using B+ trees for sorting

- Scenario: Table to be sorted has a B+ tree index on sorting attribute(s).
  - Retrieve students in increasing order of *age*.
- **Idea**: Can retrieve records in order by traversing leaf pages.
- Is this a good idea? Cases to consider:
  - B+ tree is clustered — ***Good idea!***
  - B+ tree is not clustered — ***Could be a very bad idea!***

Clustered B+ tree used for sorting

- Alternative 1: cost of retrieving all leaf pages
- Alternative 2: also cost of retrieving data records, but reading each page just once.



Almost always better than external sorting!

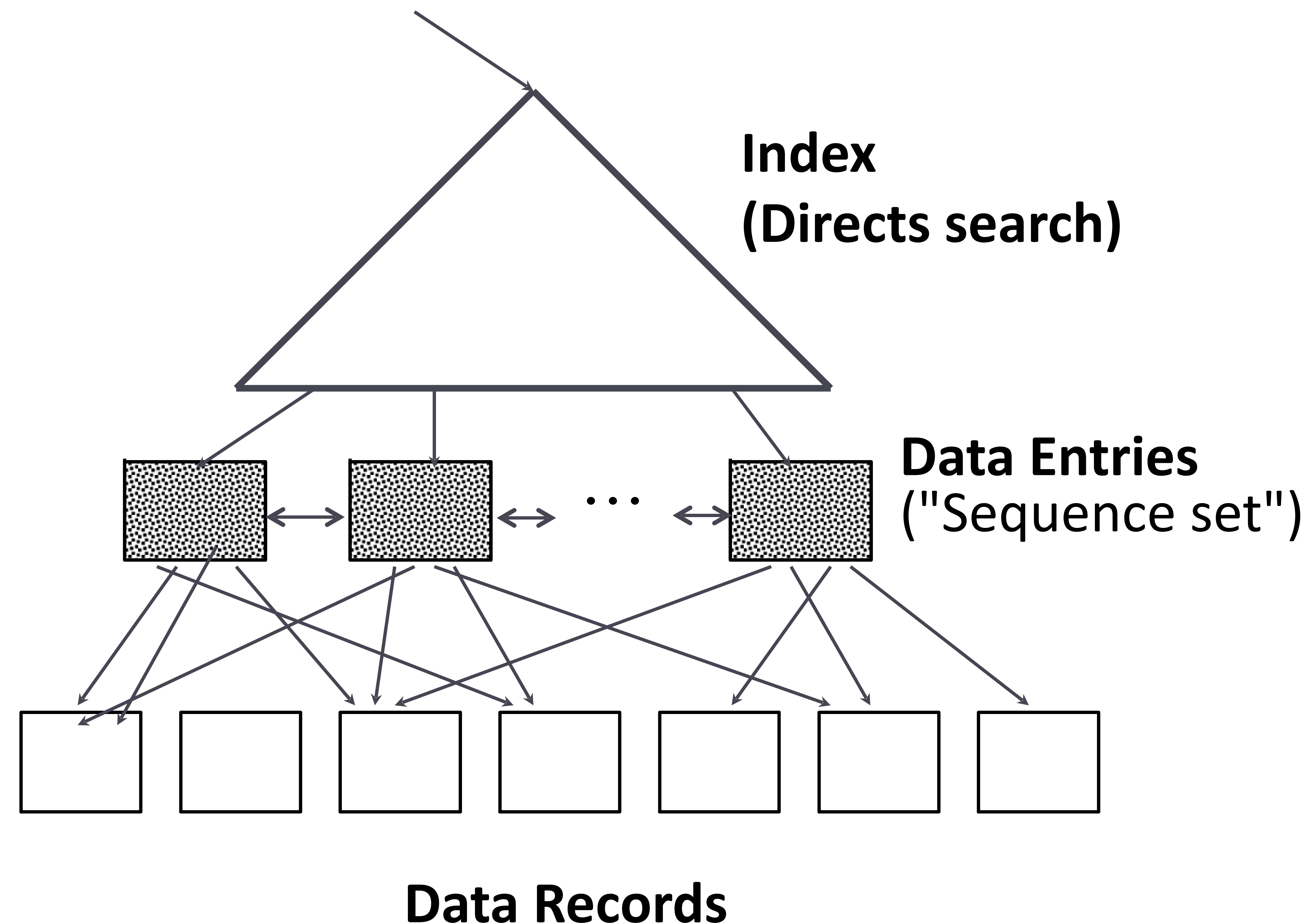
Unclustered B+ tree used for sorting

- Alternative 2: each data entry contains *rid* of a data record. In general, one I/O per data record!

Worse case I/O: RN

R: # records per page

N: # pages in file



External sorting vs. unclustered index

N	Sorting	R=1	R=10	R=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,0 00	100,000,000	1,000,000,000

For sorting

- *B*=1,000

- *R*: # of records per page
  R=100 is the more realistic value.
- *Worse case numbers (R, N) here*!

Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

Equality joins with one join column

```
SELECT *
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
```

semantics

selection (σ) cross product (x)

too expensive!

Schema for examples

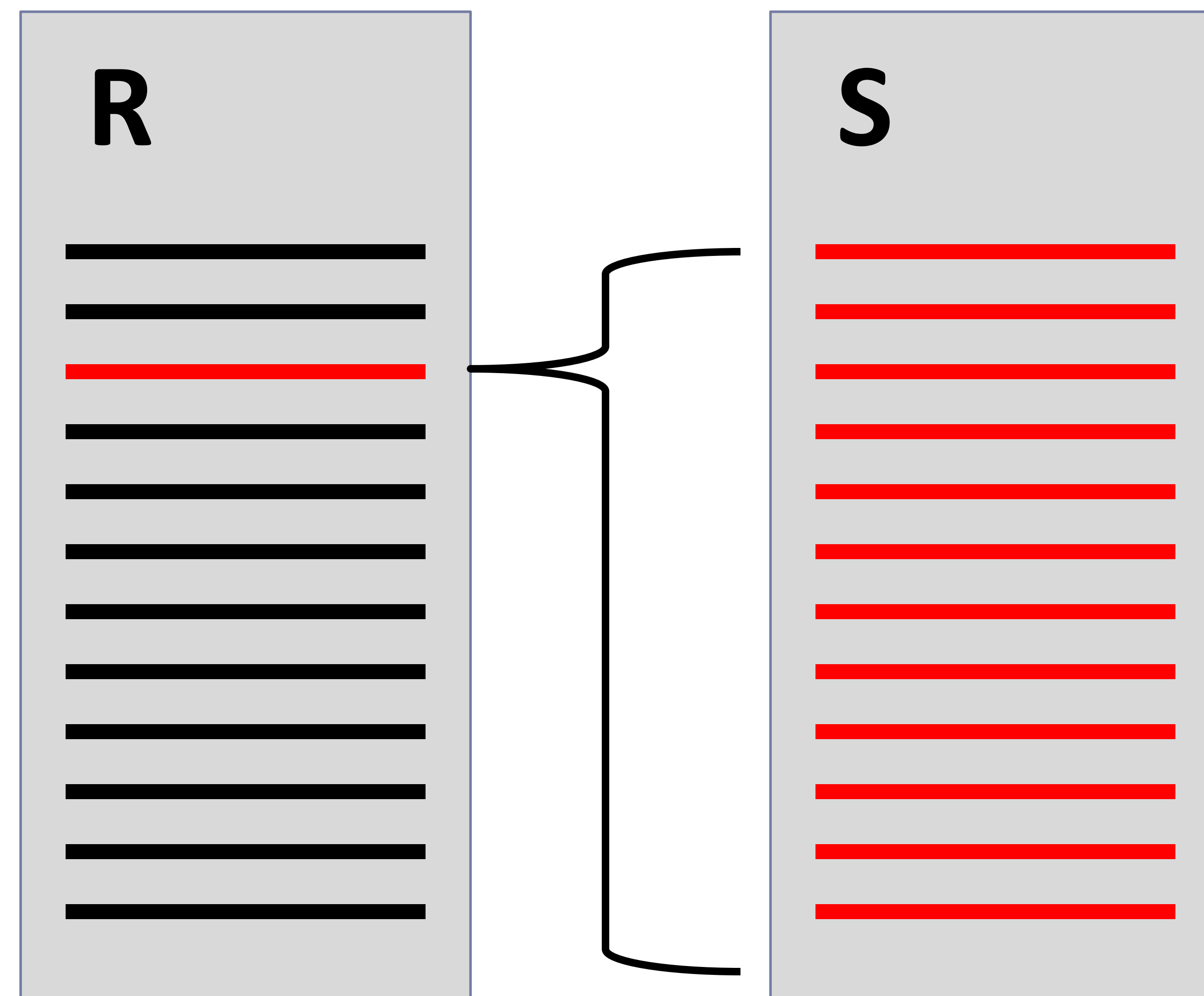
Sailors (<u>*sid*: integer</u>, *sname*: string, *rating*: integer, *age*: real)
Reserves (<u>*sid*: integer, *bid*: integer, *day*: date</u>, *rname*: string)

- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - 500 pages.
- Reserves:
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - 1000 pages.

- Cost metric: # I/Os

Simple nested loops join (NLJ)

foreach tuple r in R **do**
    foreach tuple s in S **do**
        if $r_i == s_j$ then
            add <r, s> to result



For each tuple in R, read all of S

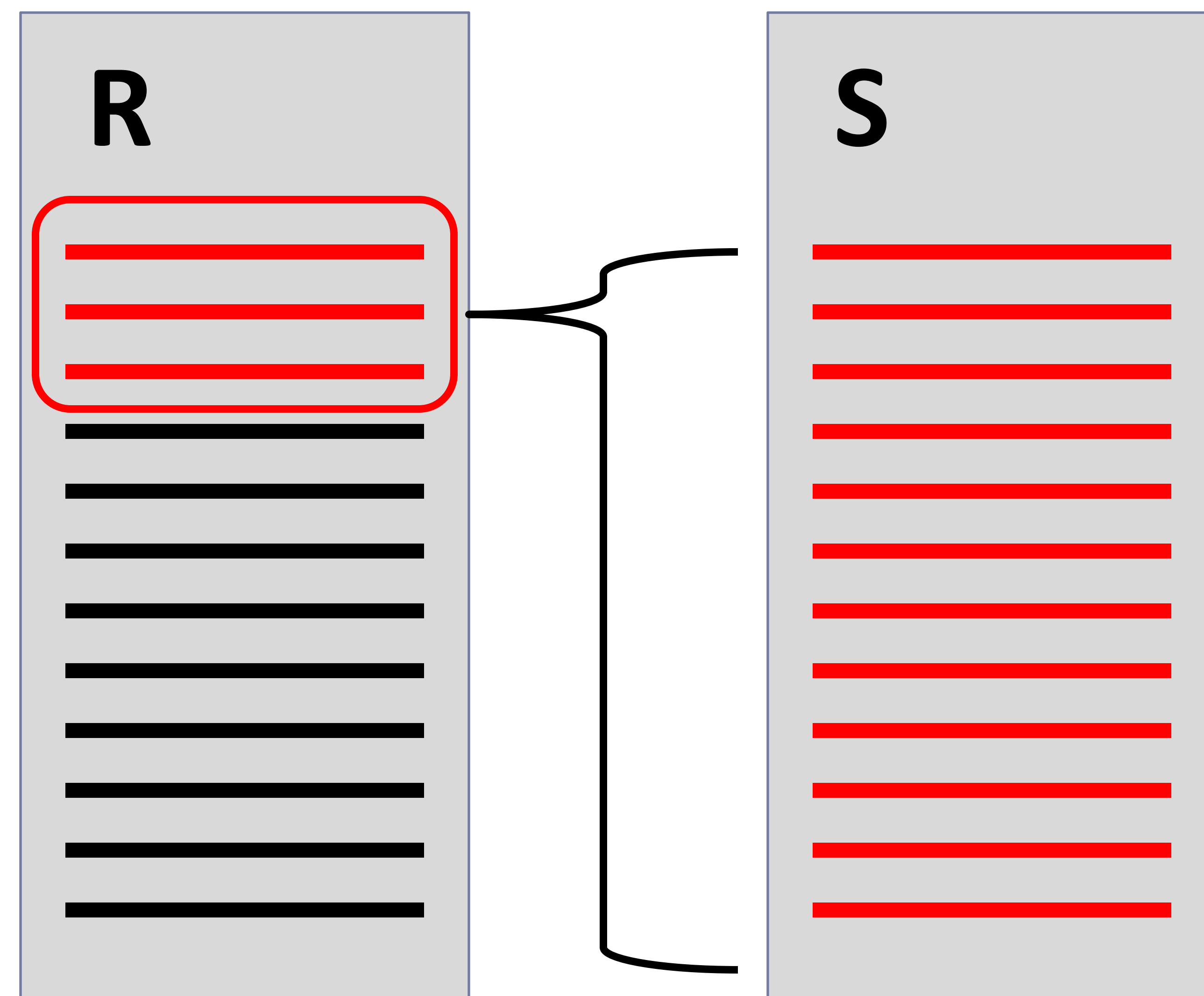
Cost: $M + (p_R\ M)\ N$ =
=1000 + 100*1000*500
= 1,000+ ($5 * 10^7$) I/Os.

assuming 10ms/IO, it will take 140 hours

Page-oriented nested loops join

- How can we improve Simple NLJ?

```
foreach page of R do
    foreach page of S do
        write out each matching pair <r, s>
        //r is in R-page, s is in S-page
```



For each page in R, read all of S

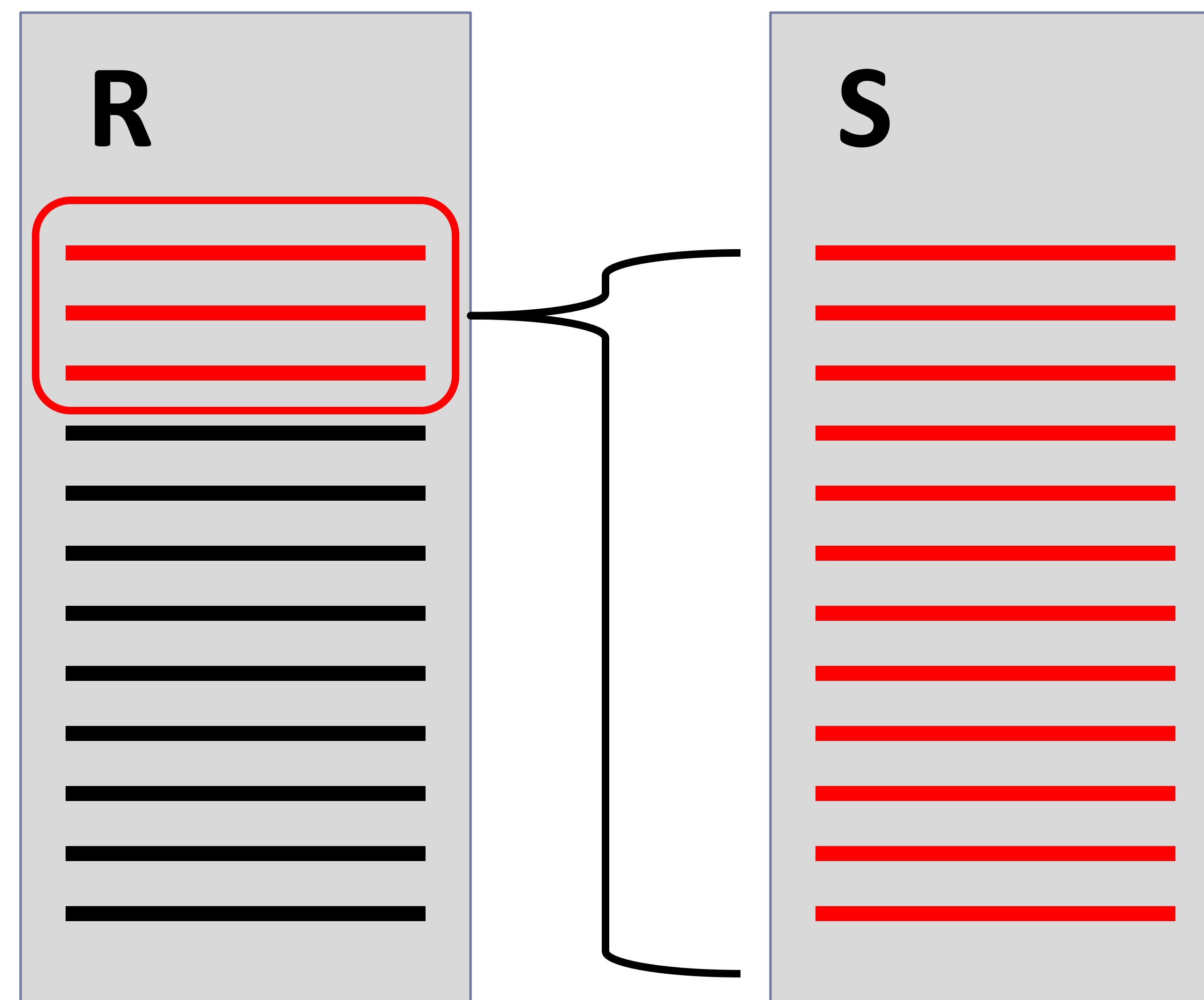
Cost: M + M N =
=1000 + 1000*500
= 501,000 I/Os.

assuming 10ms/IO, it will take 1.4 hours

Page-oriented nested loops join

- How can we improve Simple NLJ?

```
foreach page of R do
        foreach page of S do
                write out each matching pair <r, s>
                //r is in R-page, s is in S-page
```



Which should be the outer?

How many buffers do we need?

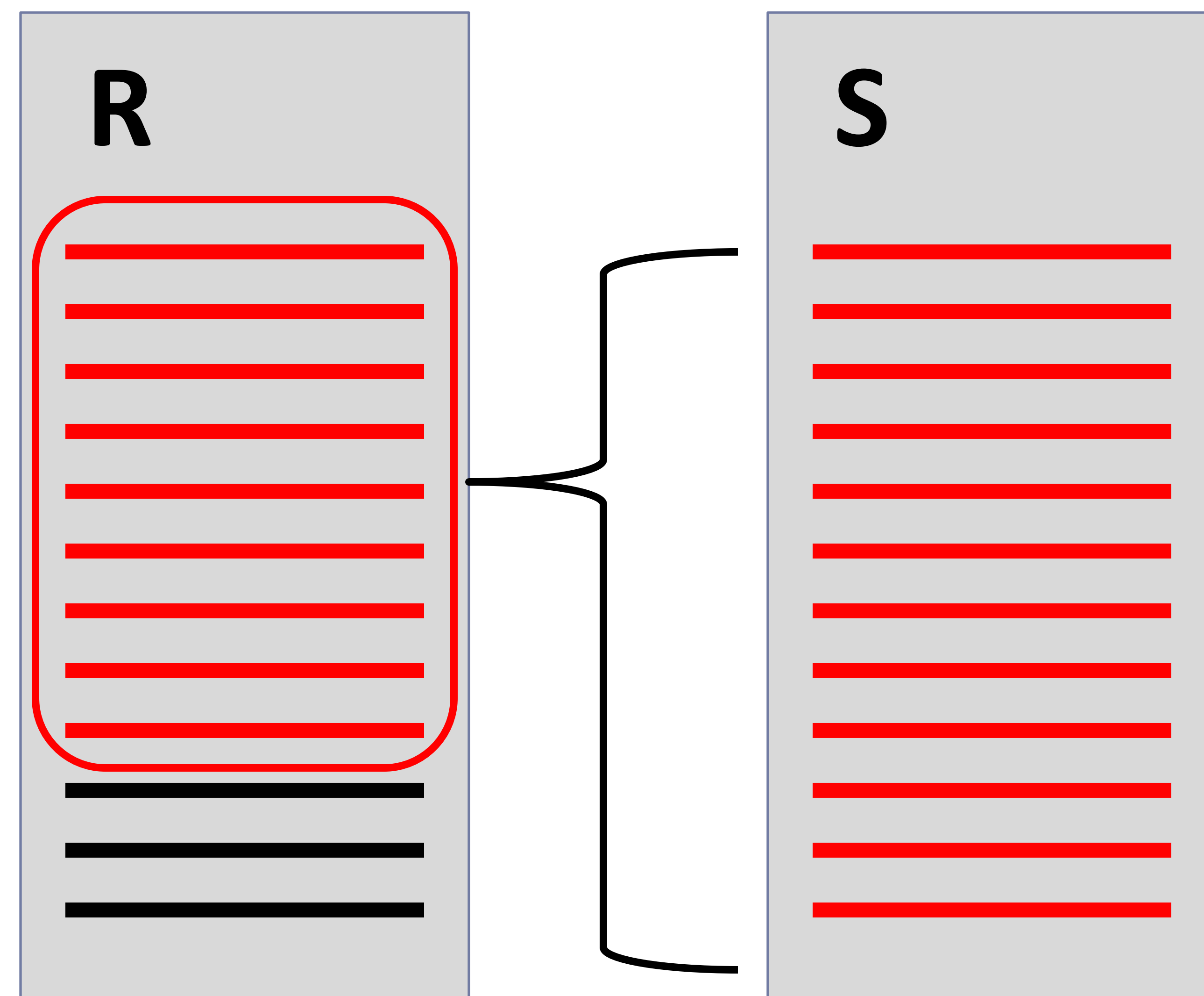
Block nested loops join

- How can we utilize additional buffer pages?
  - If the smaller reln, say R, fits in memory, use R as outer, read the inner S only once.
  - Otherwise, read a big chunk of R each time, hence reducing # times of reading S.

- Block Nested Loops Join:
  - The smaller reln R as outer, the other S as inner.
  - Buffer allocation:
    - 1 buffer for scanning the inner S
    - 1 buffer for output
    - All remaining buffers for holding a "block" of outer R

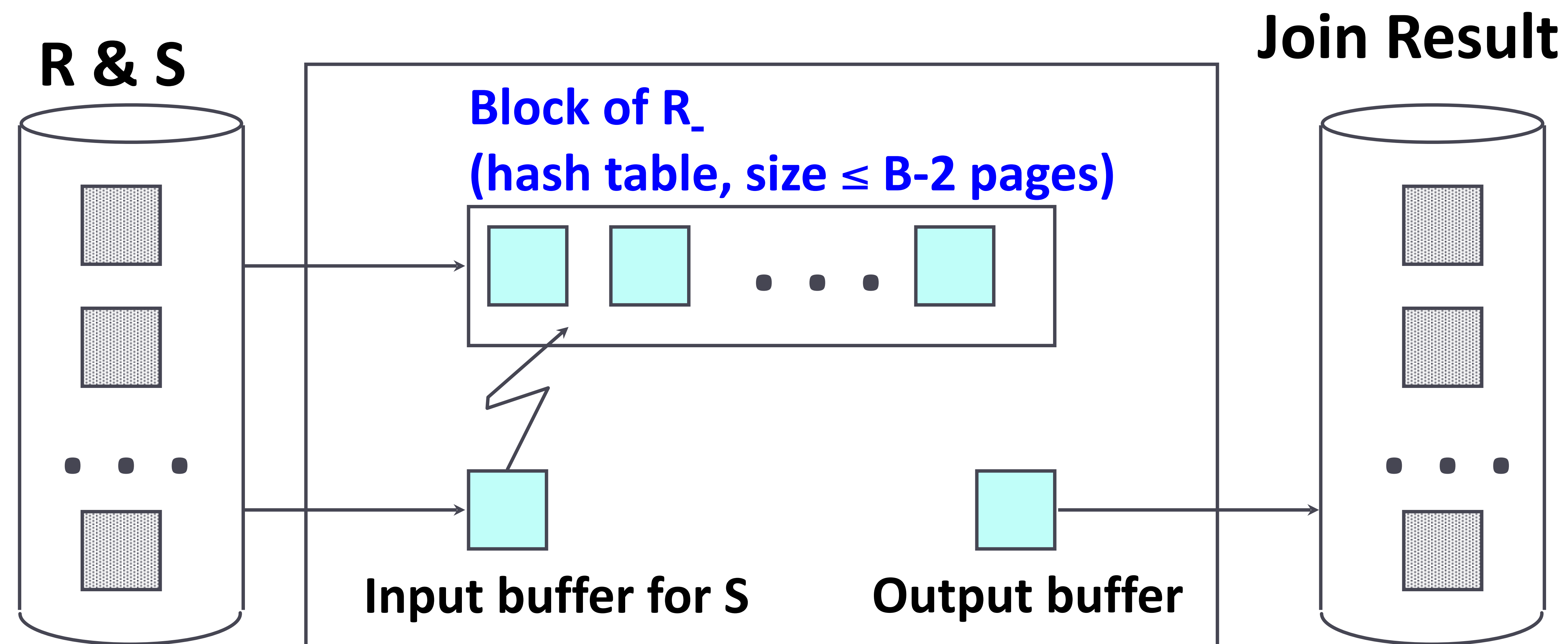
Block nested loops join

```
foreach block in R do
    foreach page in S do
        foreach matching tuple r in R-block, s in S-page do
            add <r, s> to result
```



Block nested loops join

foreach *block in R* **do**
  build a hash table on R-block
  foreach *page in S* **do**
    foreach matching tuple r in R-block, s in S-page **do**
      add <r, s> to result



Cost of block nested loops join

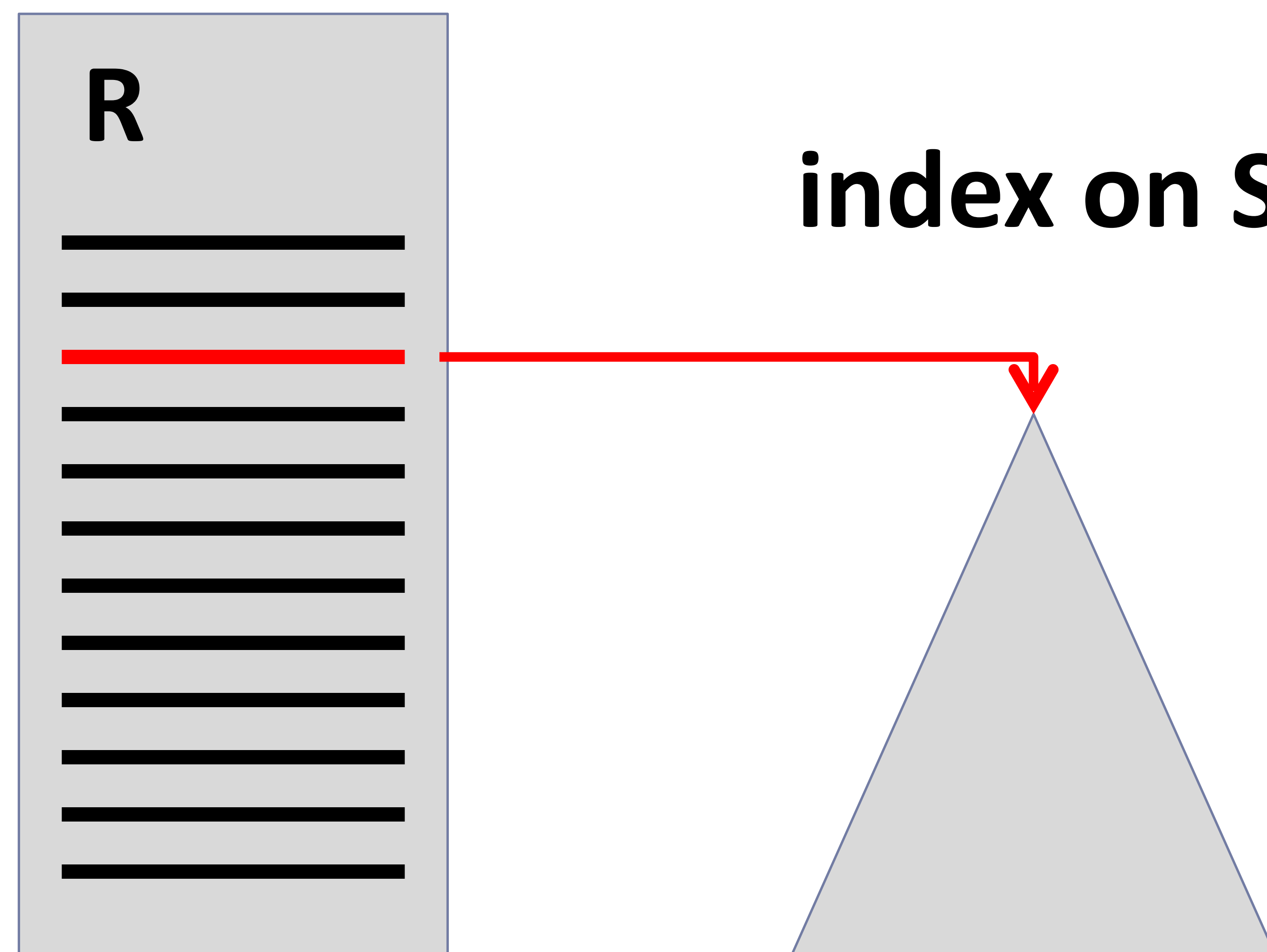
- Cost: Scan of outer + #outer blocks * scan of inner
  - B buffer pages available
  - Cost = size of outer + ⌈size of outer / B-2⌉ * size of inner

- E.g. B=102, Sailors S = 500 pages, Reserves R = 1000 pages.
  - What is the cost if S is outer, R is inner?
    - A block = B-2 = 100 pages
    - Cost = 500 + ⌈500/100⌉ * 1000 = 5,500 I/Os.
  - What is the cost if we swap R and S?
    - Cost = 1000 + ⌈1000/100⌉ * 500 = 6,000 I/Os.

Index nested loops join

- Given an index on the join column of S:

```
foreach tuple r in R do
        foreach tuple s in S where r == s (via index lookup) do
add <r, s> to result
```



For each tuple in R, check index

Cost: $M + (p_R\ M)$ index_lookup

Index nested loops join

- Given an index on the join column of S:

```
foreach tuple r in R do
        foreach tuple s in S where r == s (via index lookup) do
add <r, s> to result
```

1) Cost of index lookup:
   - *Hash index*: ~1.2 I/O to search + pages for matches
   - *B+ tree*: 2-4 I/Os to search + extra pages for matches.
2) Cost of retrieving matching S tuples:
   - *Clustered index*: one or a few I/Os (typical).
   - *Unclustered*: up to 1 I/O per matching S tuple.

Examples of index nested loops

Sailors ⋈ Reserves

- Sailors: tuple size is 50 bytes, 80 tuples per page, 500 pages.
- Reserves: tuple size is 40 bytes, 100 tuples per page, 1000 pages.

- Hash-index (Alt. 2) on *sid* of Sailors (*primary key index*):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: # of matching Sailors tuples = 1
    - 1.2 I/Os to get data entry in index
    - 1 I/O to get the *exactly one* matching Sailors tuple.
  - Total: 1000+ 100*1000*2.2 = 221,000 I/Os.

Examples of index nested loops

Sailors ⋈ Reserves

- Sailors: tuple size is 50 bytes, 80 tuples per page, 500 pages.
- Reserves: tuple size is 40 bytes, 100 tuples per page, 1000 pages.

- Hash-index (Alt. 2) on sid of Reserves:
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: # of matching Reserves tuples = ?
    - Uniform distribution: 2.5 Reserves tuples/sailor (100,000/40,000).
    - 1.2 I/Os to find the index page with data entries.
    - Cost of retrieving the tuples is 1 or 2.5 I/Os (cluster or not).
  - Total: 500+80*500*(2.2~3.7) = 88,500~148,500 I/Os.

Sort-merge (R ⋈ S) for equi-join

- Sort R and S on join column using external sorting.
- Merge R and S on join column, output result tuples.

Repeat until either R or S is finished:

- *Scanning*:
  - Advance scan of R until current R-tuple >=current S tuple,
  - Advance scan of S until current S-tuple >= current R tuple;
  - Do this until current R tuple = current S tuple.
- *Matching*:
  - Match all R tuples and S tuples with same value (called R-group and S-group of the current value).
  - Output <r, s> for all pairs of such tuples.

Example of sort-merge join

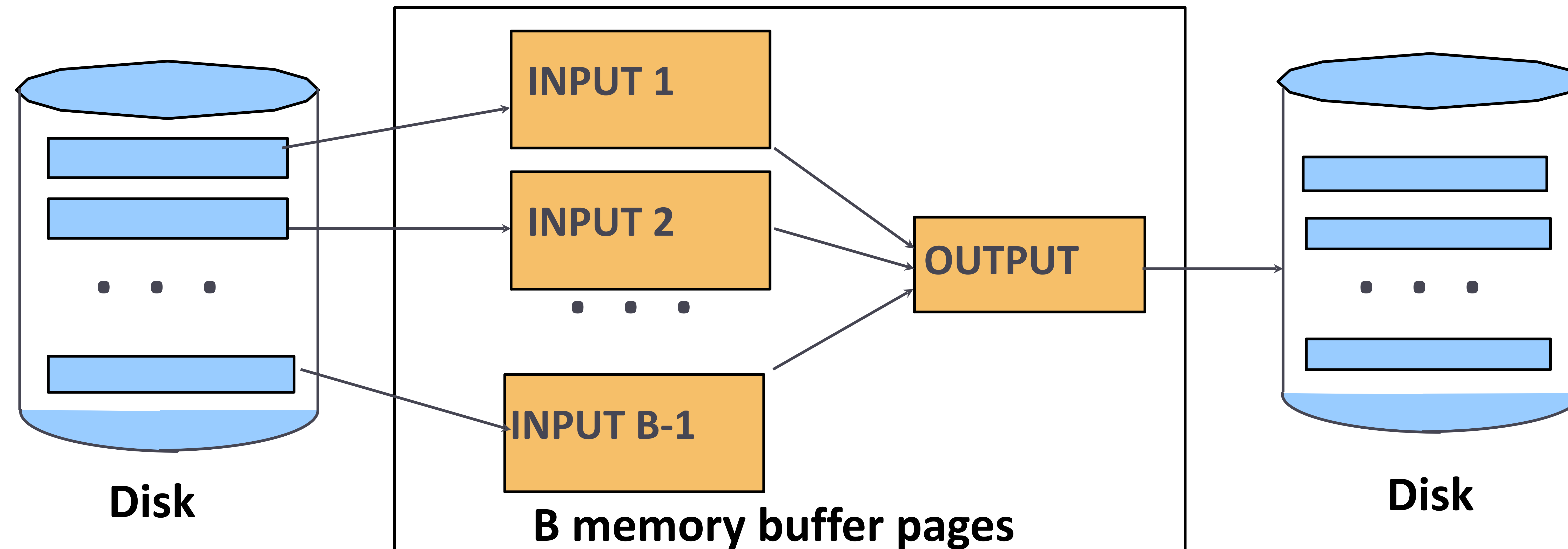
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: Sorting_cost(R) + Sorting_cost(S) + Merging_cost
  - Merging_cost ∈ [M+N, M*N]
  - M+N: *foreign key join* with the referenced reln. as inner.
  - M*N: uncommon but possible. When?

Refinement of sort-merge join

- Is there a guaranteed **linear-time** algorithm?

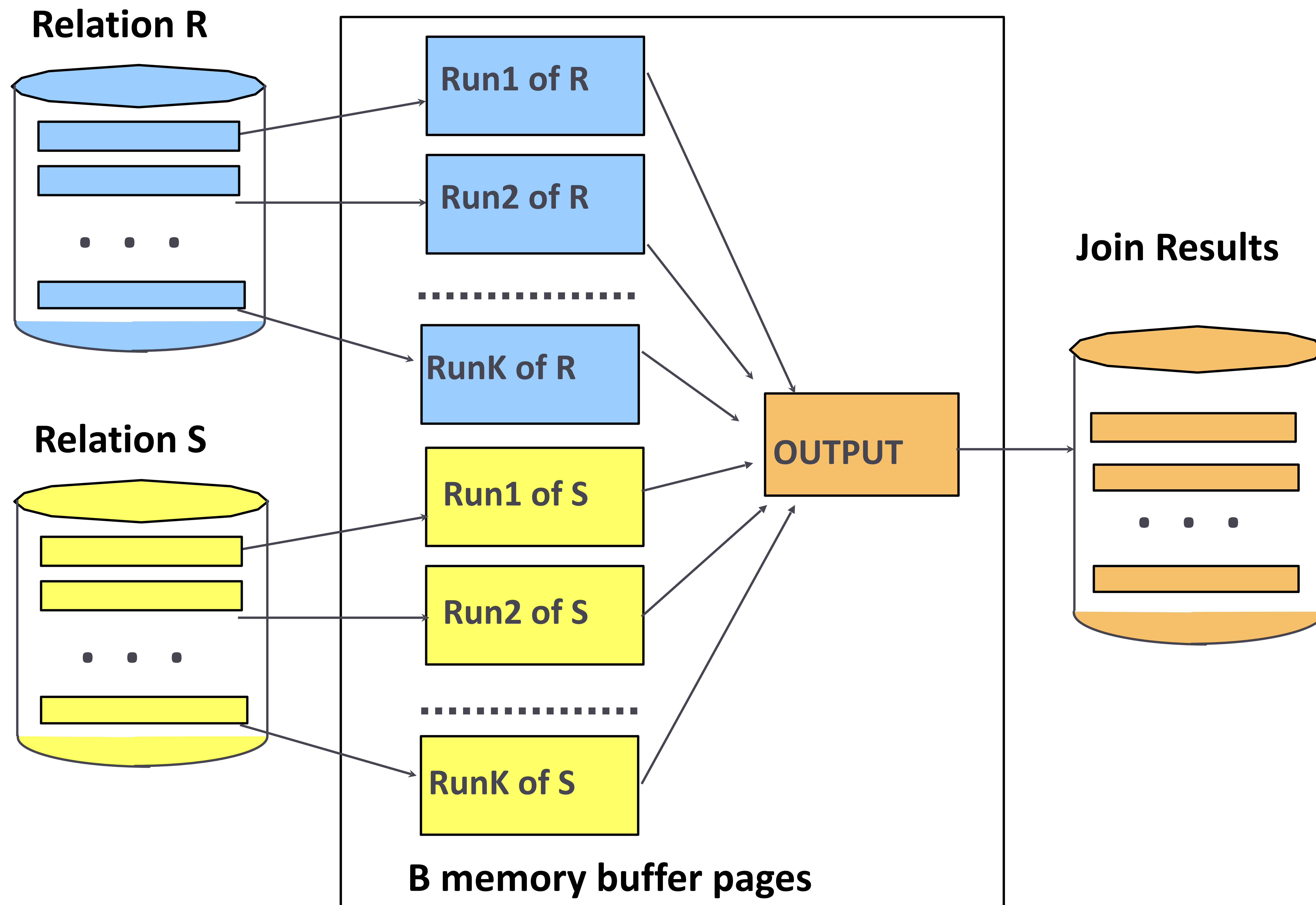


- Observed repeated merging phases:
  - *Sorting* of R and S has respective merging phases.
  - *Join* of R and S also has a merging phase.
  - Combine all these merging phases!

Two-pass sort-merge join

- Pass 1 *Sorting*: sort subfiles of R and S individually
- Pass 2 *Merging*: merge sorted runs of R and S
  - merge sorted runs of R,
  - merge sorted runs of S, and
  - compare R and S tuples using the *join condition*.
- Assume that we don't have too many duplicates…

Merging in two-pass sort-merge

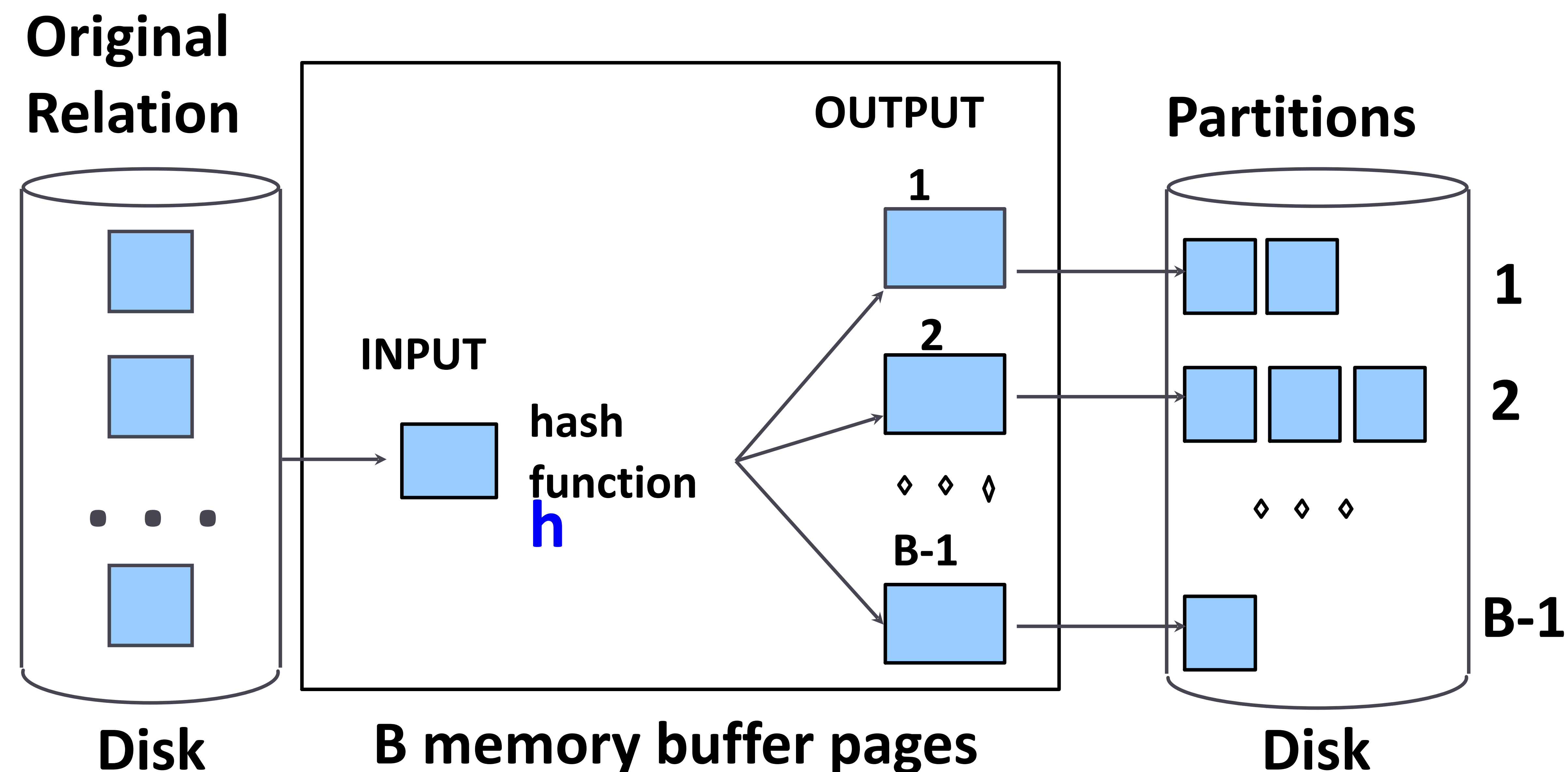


Memory requirement and cost

- Memory requirement for two-pass sort-merge:
  - Let U be the size of the *larger* relation, $U = \max(M, N)$.
  - *Sorting* pass produces sorted runs of size up to 2B. So, Number of runs per relation $\leq U/2B$.
  - *Merging* pass holds sorted runs of both relations and an output buffer. So, $2*(U/2B) + 1 \leq B \rightarrow$ **B >** $\sqrt{U}$
- Cost: read & write each relation in sorting pass
  + read each relation in merging pass
  (+ writing result tuples, ignore here) = 3 (M+N) !

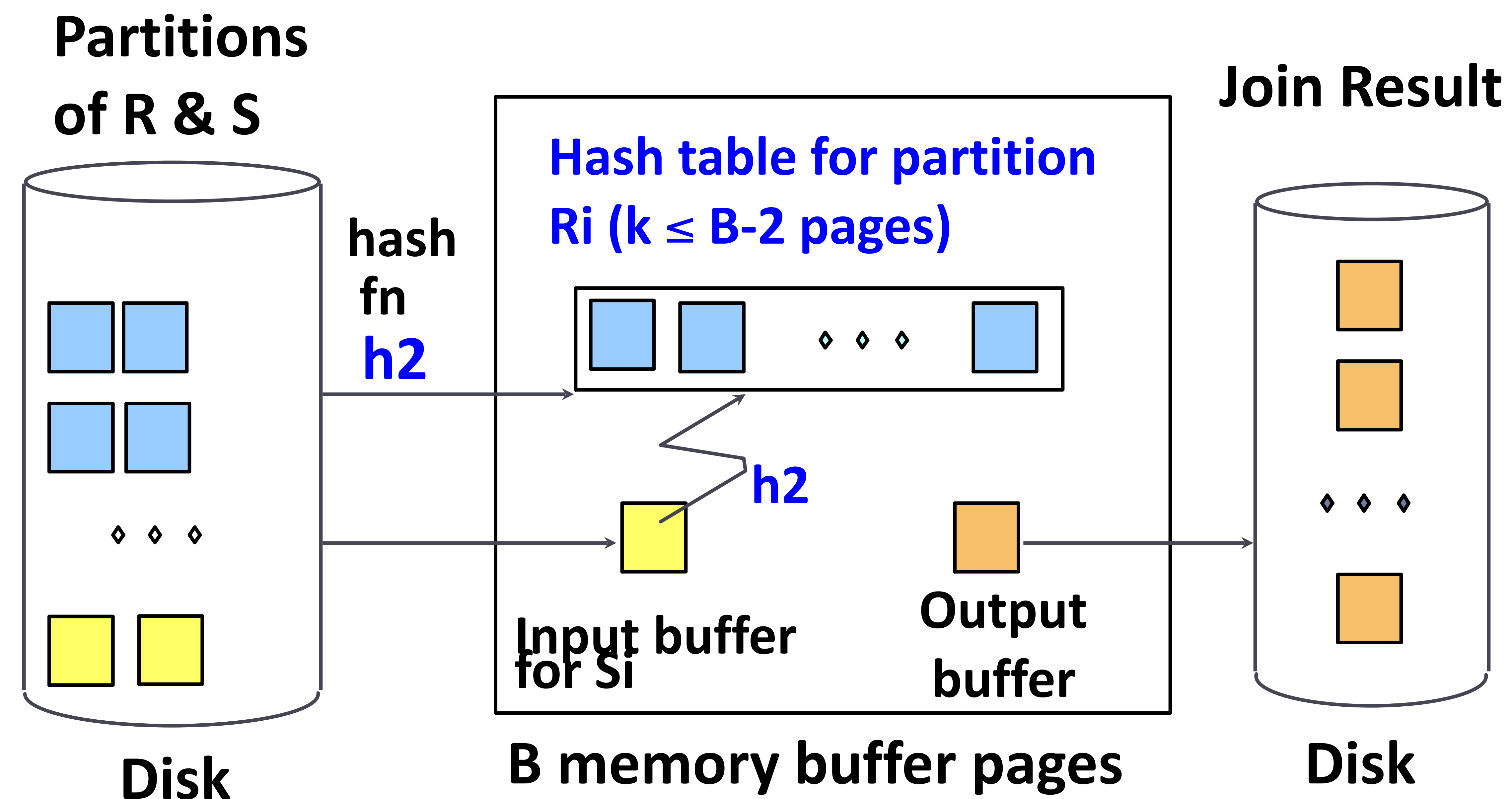
Hash-join for equi-join

- **Idea**: For an *equi-join*, partition both R and S using a hash function s.t. R tuples will only match S tuples in partition i
- Phase 1 *Partitioning*: Partition both relations using hash function **h** (Ri tuples will only match with Si tuples).



Hash-join

- Phase 2 Probing:
  - Read in partition Ri, build hash table using h2 (≠h!).
  - Scan partition Si, one page at a time, search for matches.



Memory requirement

- Partitioning: # partitions in memory $\leq$ B-1,
  Probing: to fit each Ri in memory, size of partition $\leq$ B-2.
  - A little more memory needed to build hash table, but ignored here.
- Assuming uniformly sized partitions, $L = \min(M, N)$:
  - L / (B-1) ≤ (B-2) → B 〉 $\sqrt{L}$
  - Use the *smaller* relation as the building relation in probing phase.
- What if hash fn h does not partition uniformly?
  - One or more R partitions may not fit in memory.
  - Can apply hash-join recursively to this R-partition and the corresponding S-partition. Higher cost, of course…

Cost of hash-join

- Partitioning: reads+writes both relns; 2(M+N).
  Probing: reads both relns; M+N I/Os.
  Total cost = 3(M+N).
  - In our running example, a total of 4500 I/Os using hash join (compared to 501,000 I/Os w. Page NLJ).
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os.
  - Hash Join superior on this count if relation sizes differ greatly.
    Assuming $M < N$, what if $sqrt(M) < B < sqrt(N)$?
  - Sort-Merge less sensitive to data skew; result is sorted.

General join conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - Block NL works fine.
  - For Index NL,
    - use index on ⟨*sid, sname*⟩ if available; or
    - use an index on *sid* or *sname*, check the other predicate on the fly.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

General join conditions

- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need B+ tree index.
    - Range probes on inner; number of matches likely to be much higher than for equality joins.
    - Clustered index is much preferred.
  - Block NL often works well.
  - Hash Join, Sort Merge Join not applicable.

Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

The projection operation

```
SELECT   DISTINCT R.sid, R.bid
FROM     Reserves R
```

- Projection consists of two steps:
  - Remove attributes not in the projection list.
  - If DISTINCT is specified, eliminate any duplicate tuples.
- Algorithms: *single-relation sorting* and *hashing* based on *all* remaining attributes.

Projection based on sorting

- *Sorting pass*: modified to remove unwanted fields.
  - Runs of about 2B pages are produced.
  - But tuples in runs are smaller than input tuples. (Size ratio depends on # and size of fields that are dropped.)

- *Merging passes*: modified to eliminate duplicates.
  - # result tuples smaller than input (depends on # of duplicates.)

Projection based on hashing

- *Partitioning phase*: Partition input relation using *h1*; for each tuple, discard unwanted fields.
  - Result is B-1 partitions (of tuples with only wanted fields). 2 tuples from different partitions guaranteed to be distinct.

- *Duplicate elimination phase*: Read each partition, build an in-memory hash table using *h2* on *all* fields, discard duplicates.

Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

Intersection

- Intersection: Tuples in both reln. 1 and reln. 2.

Equality join on *all* fields!

Union, Set Difference

- Union: Tuples in either reln. 1 or reln. 2.
  - Selects *distinct* values only.
- Set-difference: Tuples in reln. 1, but not in reln. 2.
- Both can be implemented using *two-relation sorting* or *hashing*.
  - Details see the textbook…

Outline

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

Aggregate operations (AVG, MIN, etc.)

```
SELECT min(S.age)
FROM   Sailors S
WHERE  S.rating = 10
```

- Aggregation without grouping
  - *File scan*: in general, requires scanning the relation.
  - *Index only scan*: if a tree index's search key includes all attributes in the SELECT and WHERE clauses.
    - e.g. B+tree on ⟨rating, age⟩

Aggregate operations

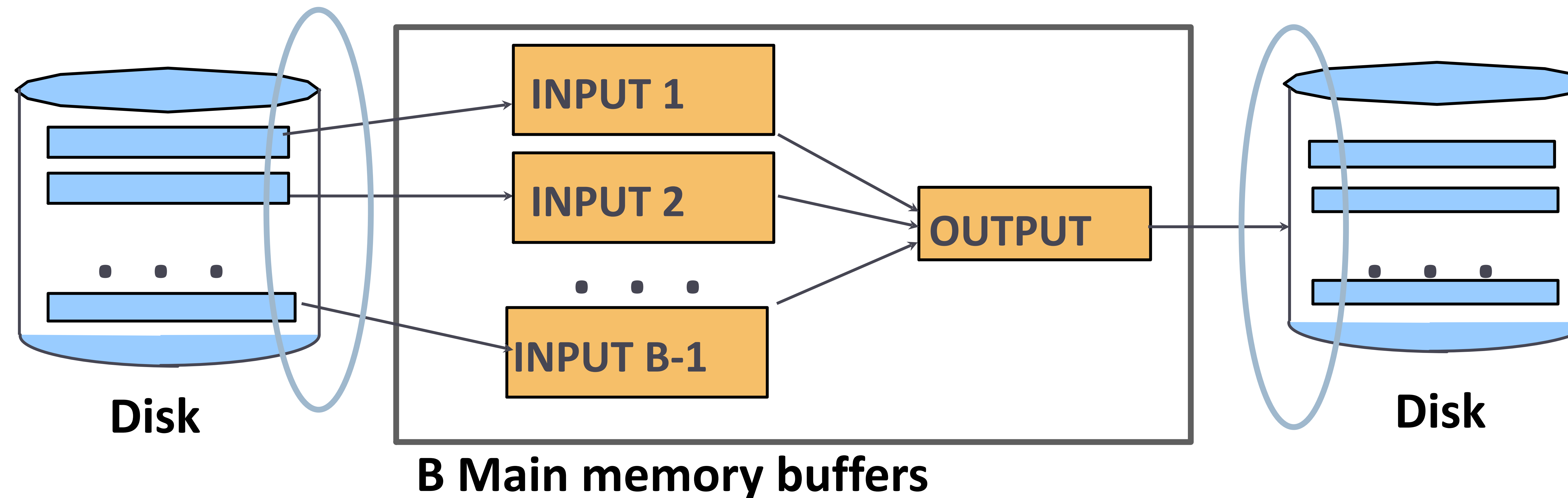
```
SELECT   min(S.age)
FROM     Sailors S
WHERE    S.rating > 5
GROUP BY S.rating
```

- Aggregation with grouping (GROUP BY)
  - *Single-relation sorting*: sort by group-by attribute(s); compute aggregate for each group in last merging phase.
  - *Single-relation hashing*: hash on group-by attribute(s): compute aggregate using in-memory hash table for each partition.
  - *Index only scan*: if a tree index's search key includes all attributes in SELECT, WHERE and GROUP BY clauses.
    - e.g. B+tree on ⟨rating, age⟩

I/O cost versus number of I/Os

- Cost metric has so far been the number of I/Os.
- Issue 1: effect of sequential (blocked) I/O?
  - Refine external sorting using *blocked I/O*
- Issue 2: parallelism between CPU and I/O?
  - Refine external sorting using *double buffering*

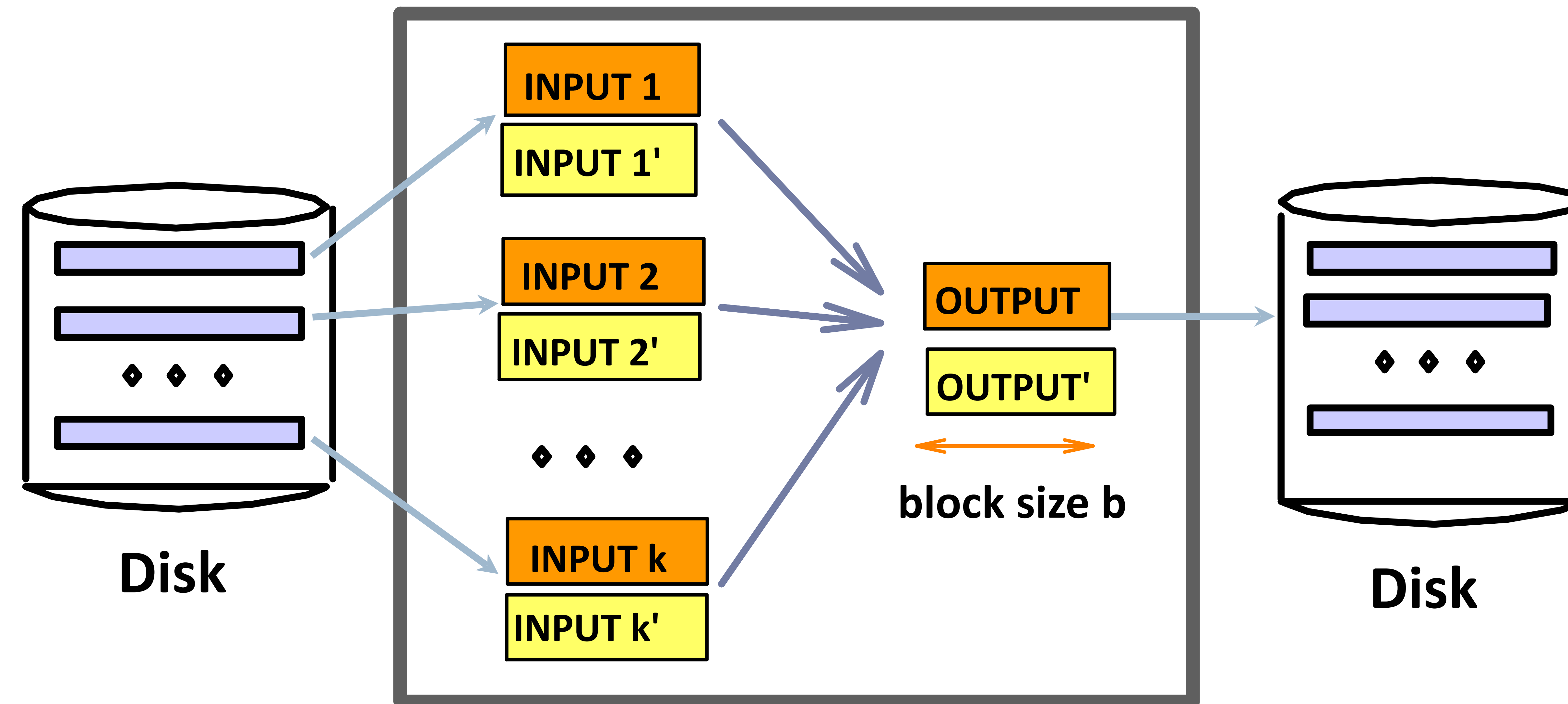
Blocked I/O for External Merge Sort



- To reduce I/O cost, make each input buffer a *block* of pages.
  - But this will reduce fan-out during merge passes! E.g. from B-1 inputs to (B-1)/2 inputs.
  - In practice, most files still sorted in 2-3 passes.

Double Buffering

- What happens when an input block has been consumed?



- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.
  - Potentially, more passes.
  - In practice, most files *still* sorted in 2-3 passes.

Summary so far

- Selections
- Sorting routine
- Joins
- Projections
- Set operators
- Group By aggregation

Different ways to perform these operations and their cost

$$N = ((z*2) + ((z*3) + y)) / x$$

Given $x = 1$, $y = 0$, and $z = 4$, solve for N

In what order did you perform the operations?

$$N = ((z*2) + ((z*3) + y)) / x$$

Given x = 1, y = 0, and z = 4, solve for N

but now assume:

* costs 10 units

\+ costs 2 units

/ costs 50 units

How will you do the operations now?

$$N = ((z*2) + ((z*3) + y)) / x$$

Algebraic laws:

1. (+) identity: $x+0 = x$
2. (/) identity: $x/1 = x$
3. (*) distributes: $(n*x+n*y) = n*(x+y)$
4. (*) commutes: $x*y = y*x$

assume:

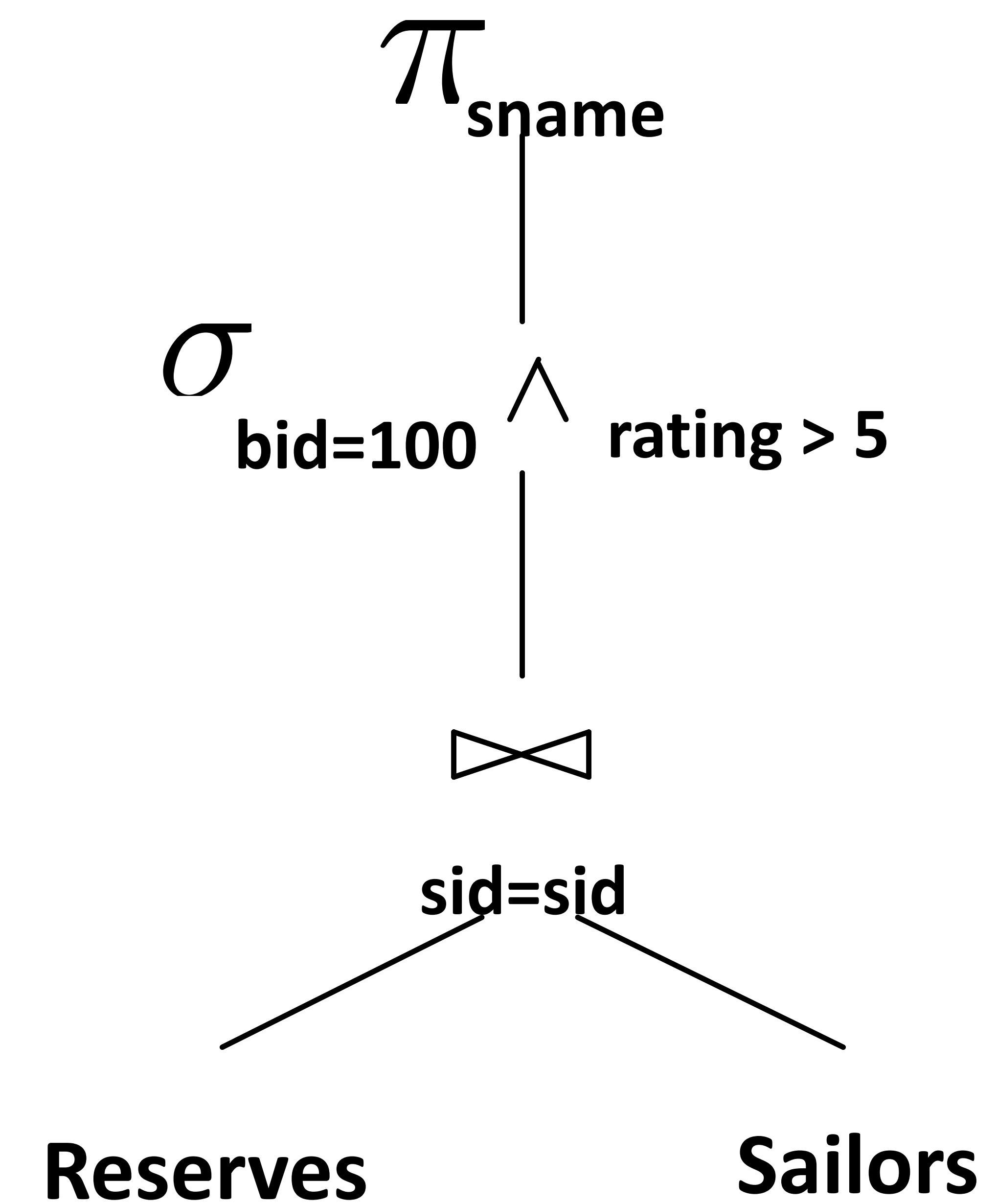
* costs 10 units
+ costs 2 units
/ costs 50 units

$$N = (2+3)*z$$

2 operations instead of 5, no division

Relational algebra tree

Relational Algebra Tree:



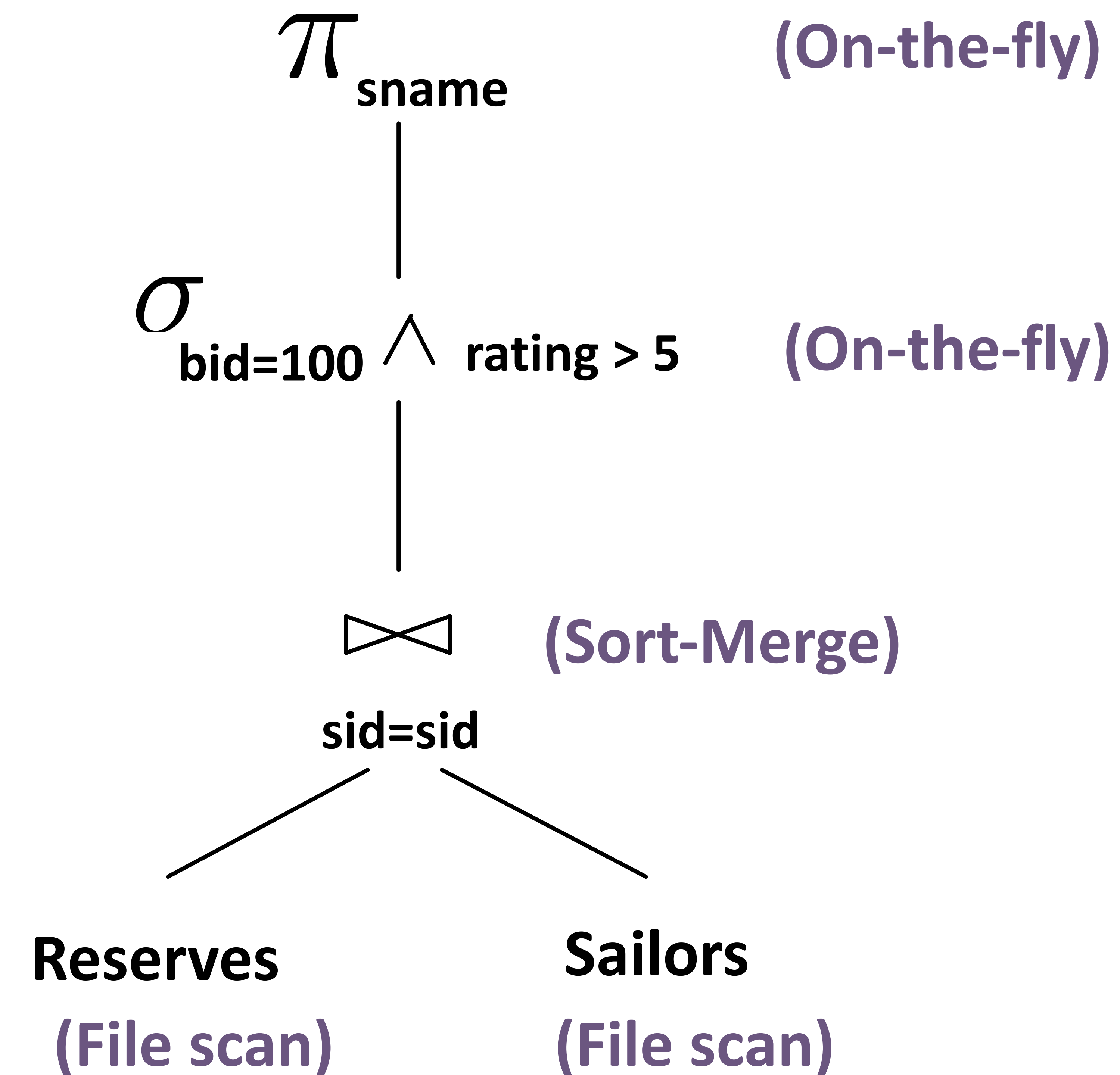

```
SELECT  S.sname
FROM    Reserves R, Sailors S
WHERE  R.sid=S.sid AND
           R.bid=100 AND S.rating>5
```

Expression in Relational Algebra (RA):

$\pi_{sname} (\sigma_{bid=100 \wedge rating>5} (\text{Reserves} \bowtie_{sid=sid} \text{Sailors}))$

Query evaluation plan

- *Query evaluation plan* extends an RA tree with:
  - *access method* for each relation;
  - *implementation method* for each other operator.



- Missed opportunities:
  - Selections could have been 'pushed' earlier.
  - More efficient joins.
  - Use of indexes.

Relational algebra equivalences

- **Selections**:

$$\sigma_{c1 \wedge \ldots \wedge cn}(R) \equiv \sigma_{c1}(\ldots \sigma_{cn}(R))$$ *(Cascade)*

$$\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$$ *(Commute)*

- **Projections**:

$$\pi_{a1}(R) \equiv \pi_{a1}(\ldots(\pi_{an}(R)))$$ *(Cascade)*

- **Joins**:

$$(R \bowtie S) \equiv (S \bowtie R)$$ *(Commute)*

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$ *(Associative)*

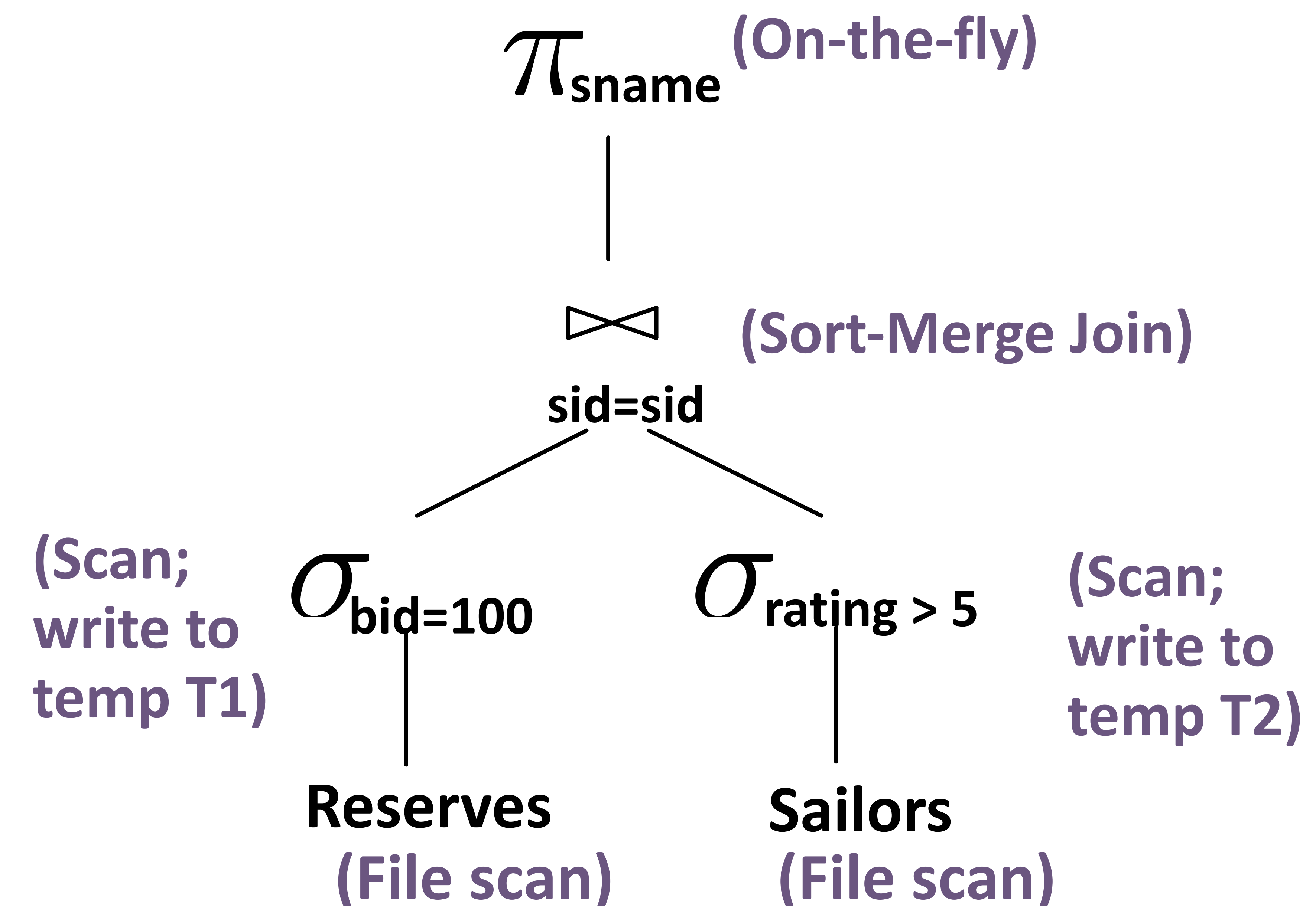
More equivalences

- $\sigma_c (R \times S) \equiv R \bowtie_c S$
- $\sigma_c (R \bowtie S) \equiv \sigma_c(R) \bowtie S$
- $\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$
  - holds if σ only uses attributes retained by π
- For $\pi_b(R \bowtie_a S)$, we can push π before $\bowtie$ by retaining only the *a* attribute and the *b* attribute (if existent)

☞ But, *aggregates* do **not** commute with other operators.

Query plan 1 (selection pushed down)

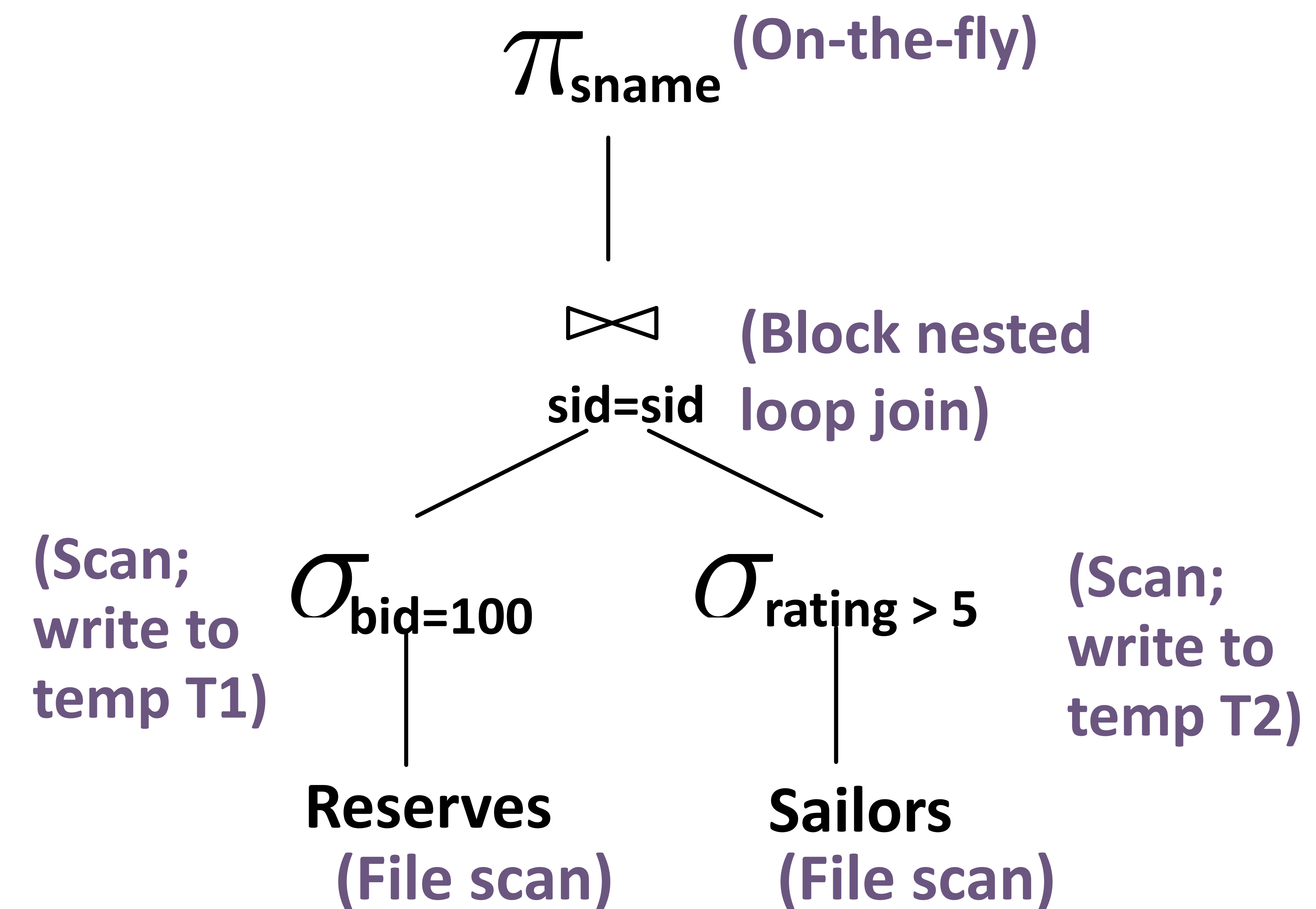
- ***Push selections below the join.***
- ***Materialization vs. Pipelining:***
  - Store a temporary relation T, if the subsequent join needs to *scan T multiple times*.
  - The opposite is *pipelining*.



- With 5 buffer pages, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages, if we have 100 boats, uniform distribution).
  - Scan Sailors (500) + write temp T2 (250 pages, if we have 10 ratings).
  - Sort-Merge join: Sort T1 (2*2*10), sort T2 (2*4*250), merge (10+250).
  - Total = 4060 page I/Os.

Query plan 2 (different join method)

- Join uses *block nested loops join* instead.



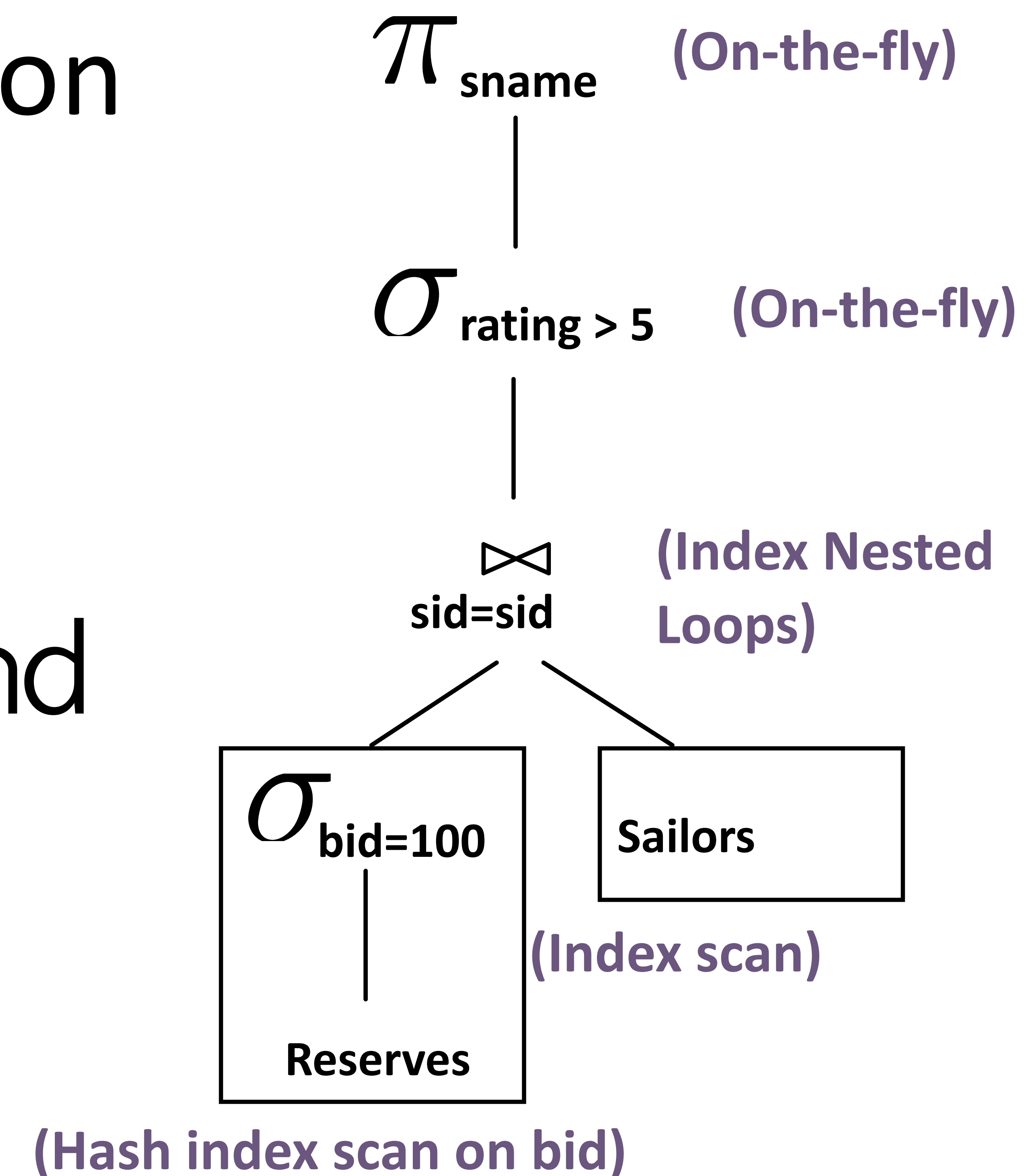
- With 5 buffer pages, cost of plan:
  - Scan Reserves (1000) + write temp T1 (10 pages).
  - Scan Sailors (500) + write temp T2 (250 pages).
  - BNL join: join cost = 10+4*250.
  - Total cost = 2770.

Using indexes

- A **tree** index *matches* (a conjunction of) terms if the attributes in the terms form a *prefix* of the index key.
  - Tree index on <*a, b, c*>
  - *a=5 AND b=3* ?
  - *a=5 AND b>6 ?*
  - *b=3 ?*
- A **hash** index *matches* (a conjunction of) terms if there is a term *attribute* = *value* for *every* attribute in the index key.
  - Hash index on <*a, b, c*>
  - *a=5 AND b=3 AND c=5 ?*
  - *a=5 AND b=3 ?*
  - *a>5 AND b=3 AND c=5 ?*

Query plan 3 (using indexes)

- *Selection using index*: clustered index on *bid* of Reserves.
  - Retrieve 100,000/100 = 1000 tuples
  - Clustering: read 1000/100 = 10 pages.
- Indexed NLJ: *pipeline* the outer and *index lookup* on sid of Sailors.
  - The outer: no need to materialize.
  - The inner: *sid* is a *key*; *at most one* match tuple, unclustered index OK.



- Cost:
  - Selection of Reserves tuples (10 I/Os).
  - For each tuple, get matching Sailors tuple (1000*(1.2+1)).
  - Total = 2210 I/Os.

Highlights of System R optimizer

- Impact： most widely used； works well for < 10 joins.
- Plan Space： too large, must be pruned.
  - Only considers the space of ***left-deep plans***.
  - Avoids cartesian products!
- Cost of a plan： approximate art at best.
  - Uses statistics to estimate ***cost of an operation*** and its ***result size***.
  - Considers a combination of CPU and I/O costs.
- Plan Search： dynamic programming
  - Prunes useless subplans.

(1) Plan space

The plans considered are:

- All *access methods*, for each reln in the FROM clause.
- All *left-deep join trees*: all the ways to join the relns one-at-a-time, with the inner reln in the FROM clause.
  - All permutations of N relns: **N factorial**!

Left-deep

D
C
A B

Bushy

A B C D

Bushy

D
C
A B

(1) Plan space

The plans considered are:

- All access methods, for each reln in FROM clause.
- All left-deep join trees: all the ways to join the relns one-at-a-time, with the inner reln in the FROM clause.
  - All permutations of N relns: N factorial!
  - But avoid cartesian products!
  - R.a = S.a and R.b = T.b, how many left-deep trees?
- All join methods, for each join in the tree.
- Appropriate places for selections and projections.
  - Not all selections can be pushed before joins.

(2) Cost estimation

- For each plan considered, must estimate its cost.
- Estimate *cost* of each operation in a plan tree:
  - Depends on input cardinalities.
  - Depends on the method (sequential scan, index scan, join method…)
- Estimate *size of result* for each operation in tree:
  - Use statistics about input relations.

Statistics in system catalog

- Statistics about each relation (R) and index (I):
  - Relation cardinality: # tuples (NTuples) in *R*
  - Relation size: # pages (NPages) in *R*

  - Index cardinality: # distinct values (NKeys) in *I*
  - Index size: # pages (INPages) in *I*
  - Index height: # nonleaf levels (IHeight) of *I*
  - Index range: low/high key values (Low/High) in *I*

  - Number of distinct values in an attribute (NKeys)
  - Histogram for an attribute

Size estimation & reduction factors

```
SELECT  attribute list
FROM  relation list
WHERE  term_1 AND ... AND term_k
```

- *Reduction factor (RF) or Selectivity* of each *term* reflects the impact of the *term* in reducing result size.
  - Assumption 1: *uniform distribution of the values.*
  - Term *col=value*: RF = *1/NKeys(I), if there is an index I on col.*
  - Term *col>value*: RF = *(High(I)-value)/(High(I)-Low(I))*
  - Term *col1=col2*: RF = *1/MAX(NKeys(I1), NKeys(I2))*
- *Result cardinality* = Max # tuples * product of all RFs.
  - Max # tuples = the product of the cardinalities of relations in the FROM clause.
  - Assumption 2: *terms are independent.*

Issue with assumption 1

- "Uniform distribution of values": often causes highly inaccurate estimates
  - E.g., distribution of gender: male (15), female (4), other(1)
  - E.g., distribution of age:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	3	3	1	2	1	3	8	4	2	0	1	2	4	9

Nkeys = 15, count = 45.

Reduction factor of '*age=14*' : 1/15? 9/45=1/5 !

- *Histogram*: approximates a data distribution

Equiwidth histograms

Equiwidth Histogram: buckets of equal size.

Frequency	8/3	4/3	15/3	3/3	15/3
Counts	8	4	15	3	15
Buckets	0 1 2	3 4 5	6 7 8	9 10 11	12 13 14

Still not accurate for value 14 with true frequency 9

Equidepth histograms

Equidepth: roughly equal counts in buckets.

Small errors for infrequent items: tolerable.

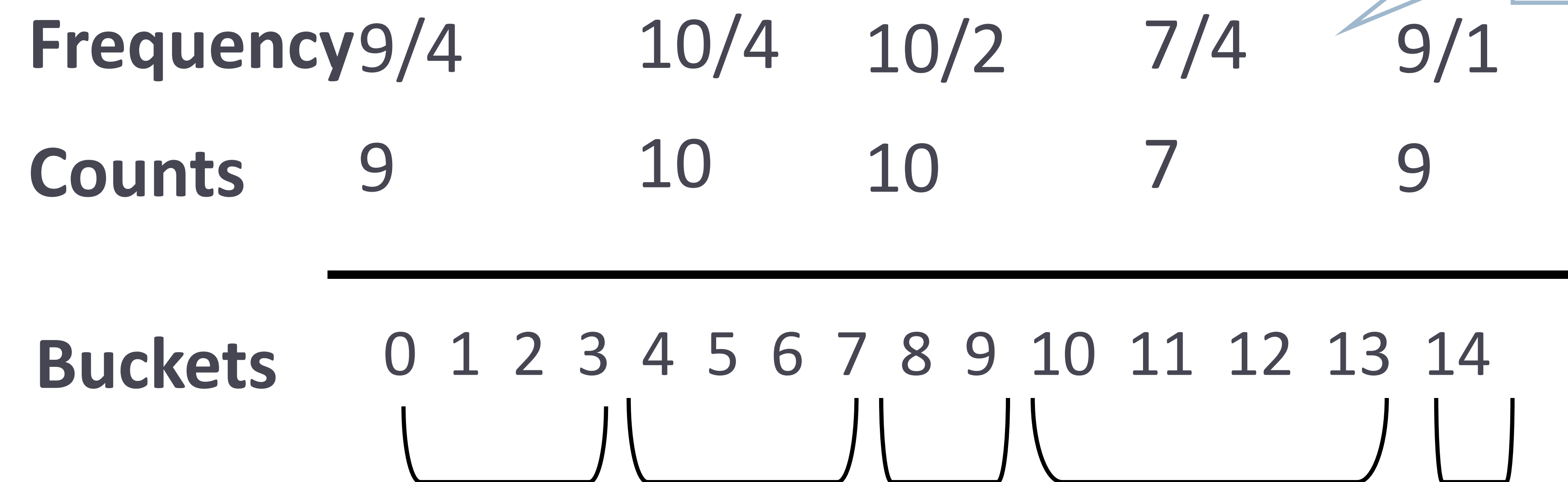
Frequency	9/4	10/4	10/2	7/4	9/1
Counts	9	10	10	7	9
Buckets	0 1 2 3	4 5 6 7	8 9	10 11 12 13	14

Now accurate for value 14: 9

- Favors *frequent* values.
- What if we have 100 distinct values and 20 frequent values?
  - Keep a list of (MCVs) with their frequencies.
  - The histogram excludes MCVs.

Equidepth histograms

Equidepth: roughly equal counts in buckets.



- Implementation is often sampling-based:
  - Boundaries of 5 buckets {0, 4, 8, 10, 14, 14}
  - Count of tuples for each bucket (optional)
  - Number of distinct values for each bucket (optional)

Issue with assumption 2

- Independence of predicates: often causes inaccurate estimates
  - E.g., Car DB: 10 makes, 100 models.
  - RF of make='honda' and model='civic'
  - If independent, 1/10 * 1/100. In practice, much higher!
- Multi-dimensional histograms [PI'97, MVW'98, GKT'00]
  - Maintain counts and frequency in multi-attribute space.
- Dependency-based histograms [DGR'01]
  - Learn dependency between attributes and compute conditional probability P(model='civic' | make='honda')
  - Can use graphical models…

(3) Queries over a single relation

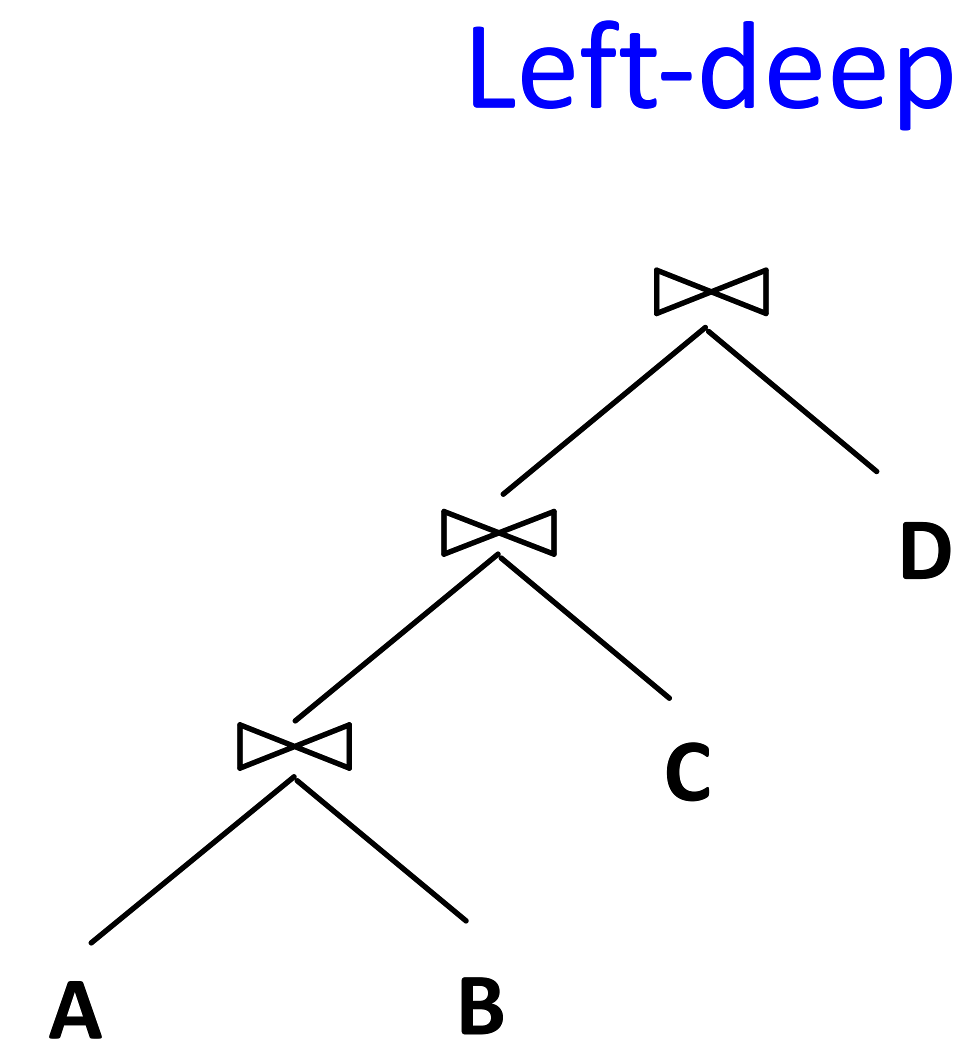
- Query involves selection, projection, and aggregation.
- Enumeration of alternative plans:

1. Each *available access path* (file/index scan) is considered, the one with least estimated cost is chosen.
2. Various operations are often carried out together:
   - If an index is used for a selection, projection is done for each retrieved tuple.
   - If no GROUP BY, the resulting tuples can be *pipelined* into the aggregate computation.
   - Otherwise, hashing or sorting is needed for GROUP BY.

(4) Queries over multiple relations

- As the number of joins increases, the number of alternative plans grows rapidly.

  - System R: (1) use *only left-deep join trees*, (2) avoid *cartesian products*.
  - Allow *pipelined* plans; intermediate results not written to temporary files.
  - Not all left-deep trees are fully pipelined!
    - Sort-Merge join: sorting phase
    - Two-phase hash join: partitioning phase



Plan search

- Left-deep join plans differ in:
  - the *order* of relations,
  - the *access path* for each relation, and
  - the *join method* for each join.
  - where *selections* are placed.
- Many of these plans share common prefixes, so don't enumerate all of them. This is a job for…
- Dynamic Programming

  "a method of solving problems that exhibit the properties of *overlapping subproblems* and *optimal substructure*."

Enumeration of left-deep plans

- <u>Enumerate with N passes</u> (if N relations are joined):
  - Base: Find best 1-relation plan for each relation.
  - Pass 1: Find best ways to join result of each 1-relation plan (as *outer*) to another relation. *(All 2-relation plans.)*
  - …
  - Pass N-1: Find best ways to join result of a (N-1)-relation plan (as *outer*) to the N'th relation. *(All N-relation plans.)*
- <u>For each subset of relations</u>, retain only:
  - cheapest *unordered* plan, and
  - cheapest plan for each *interesting order* (order for final output or a subsequent op. using sorting) of the tuples.

Enumeration of plans (Contd.)

- A k-way ($k<N$) plan is not combined with an additional relation unless there is a join condition between them.
  - Do it until all predicates in WHERE have been used.
  - That is, avoid cartesian products if possible.
- ORDER BY, GROUP BY, aggregates etc. handled as a final step, using an `interestingly ordered' plan, or an additional sorting or hashing.

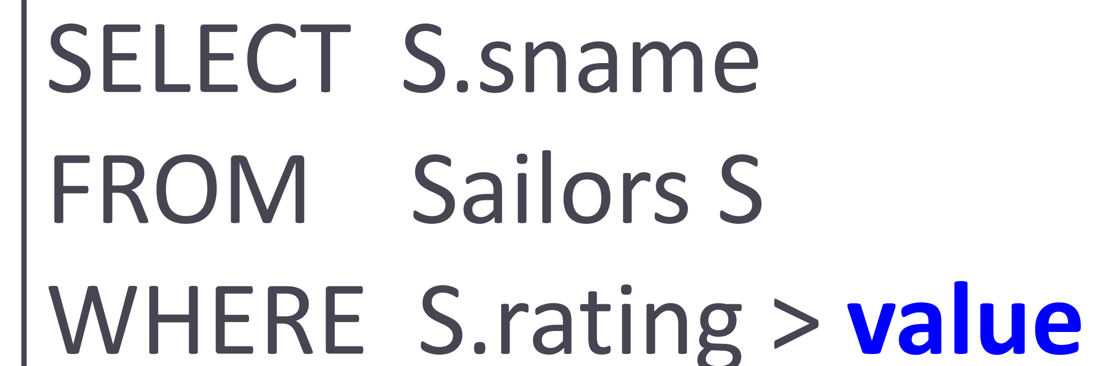
What about nested queries?

- *Nested query*: appears as an operand of a predicate in WHERE.

```
SELECT  S.sname
FROM    Sailors S
WHERE  S.rating >
  (SELECT  Avg(rating)
   FROM  Sailors)
```

- *Nested query with no correlation*: does not contain a reference to the tuple from the outer.
  - A nested query needs to be evaluated *only once*.
  - The optimizer arranges it to be evaluated before the top level query.

```
  (SELECT  Avg(rating)
   FROM  Sailors)
```

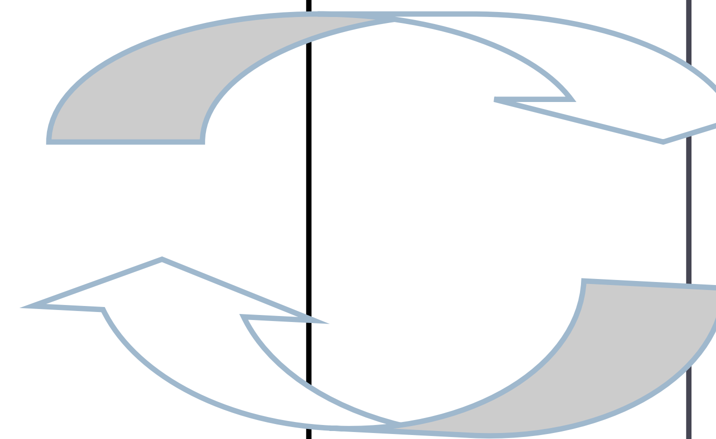


Nested queries with correlation

- Nested query with correlation: contains a reference to a tuple from the outer.
  - Nested block is optimized independently, with the value of outer tuple considered as a constant.
  - The nested block is evaluated a tuple-at-a-time.

```
SELECT  S.sname
FROM    Sailors S
WHERE  EXISTS
  (SELECT  *
   FROM    Reserves R
   WHERE  R.bid=103
   AND     R.sid=S.sid)
```

```
SELECT  S.sname
FROM    Sailors S
WHERE  EXISTS
  ( …)
```



Nested block to optimize:

```
(SELECT  *
 FROM    Reserves R
 WHERE R.bid =103
   AND   S.sid = outer value)
```

Unnesting

- ***Guideline:** Use only one "query block", if possible.*

```
SELECT DISTINCT *
FROM    Sailors S
WHERE S.sname IN
          (SELECT Y.sname
             FROM YoungSailors Y)
```

=

```
SELECT DISTINCT S.*
FROM    Sailors S,
             YoungSailors Y
WHERE S.sname = Y.sname
```

- *Not always possible ...*

```
SELECT *
FROM    Sailors S
WHERE S.sname IN
          (SELECT DISTINCT Y.sname
           FROM YoungSailors Y)
```

≠

```
SELECT S.*
FROM    Sailors S,
             YoungSailors Y
WHERE S.sname = Y.sname
```