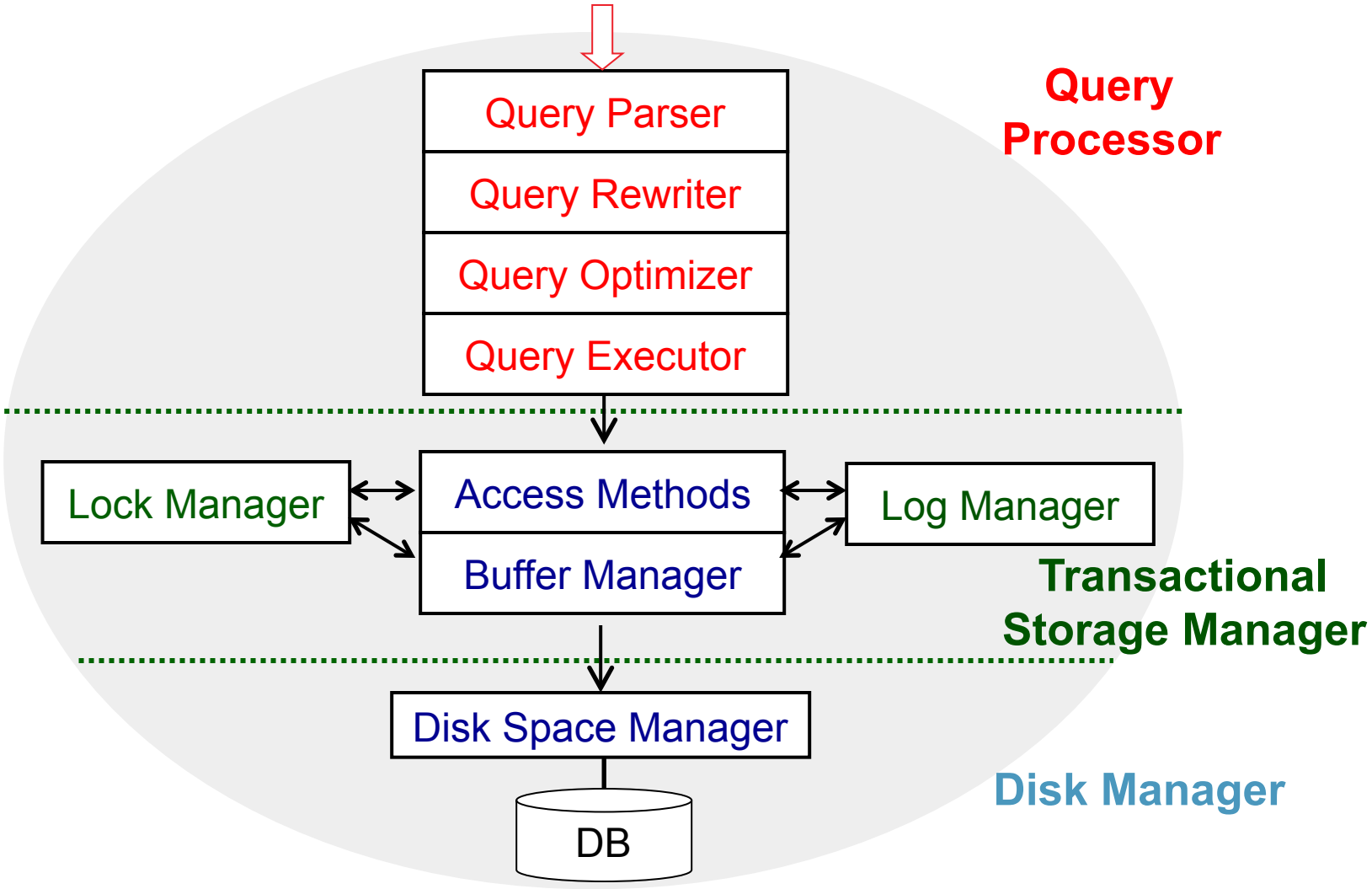
Evaluation of Relational Operations

Yanlei Diao

Slides Courtesy of R. Ramakrishnan and J. Gehrke

Overview of Query Processing



Relational Operations

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

  - *Union* ($\cup$) Tuples in either reln. 1 or reln. 2.
  - *Intersection* ($\cap$) 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
```

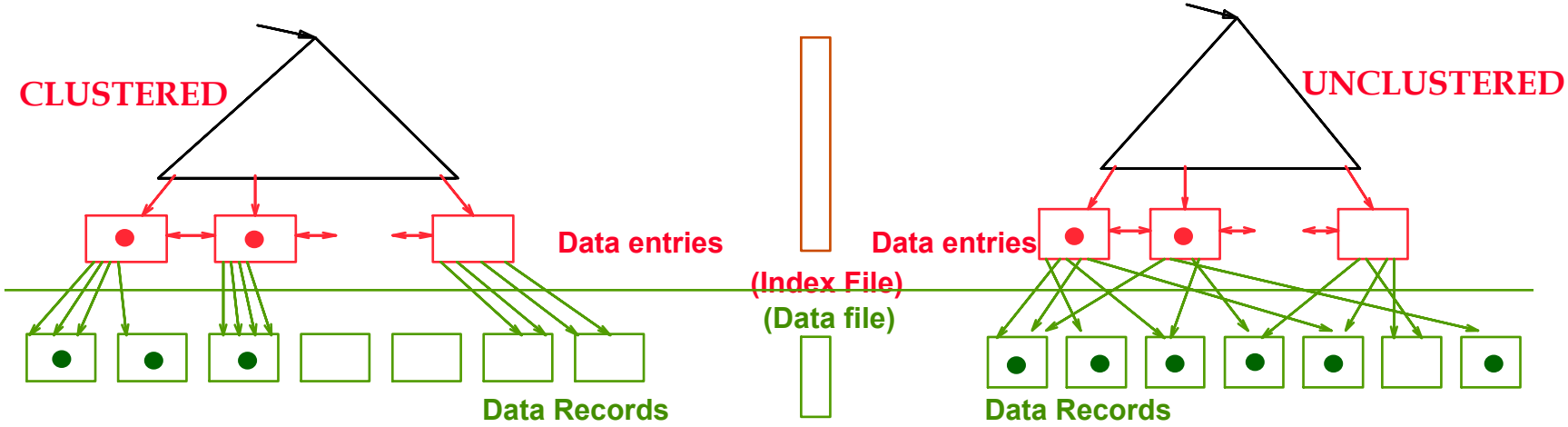
- Cost of selection includes:
  1) top down search in the index
  2) scan the relevant leaf nodes
  3) retrieve records from file (could be large w/o clustering).
- **Step 1) top down search**: ≤ 3-4 I/Os, depending on the height of the tree and buffer management

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 (contd.)



- 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!
    - Unclustered index + Sorting based on rid: ≤ 500 I/Os.
      (Bitmap Index Scan + Bitmap Heap Scan in PostgreSQL)

Outline

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

Why Sort?

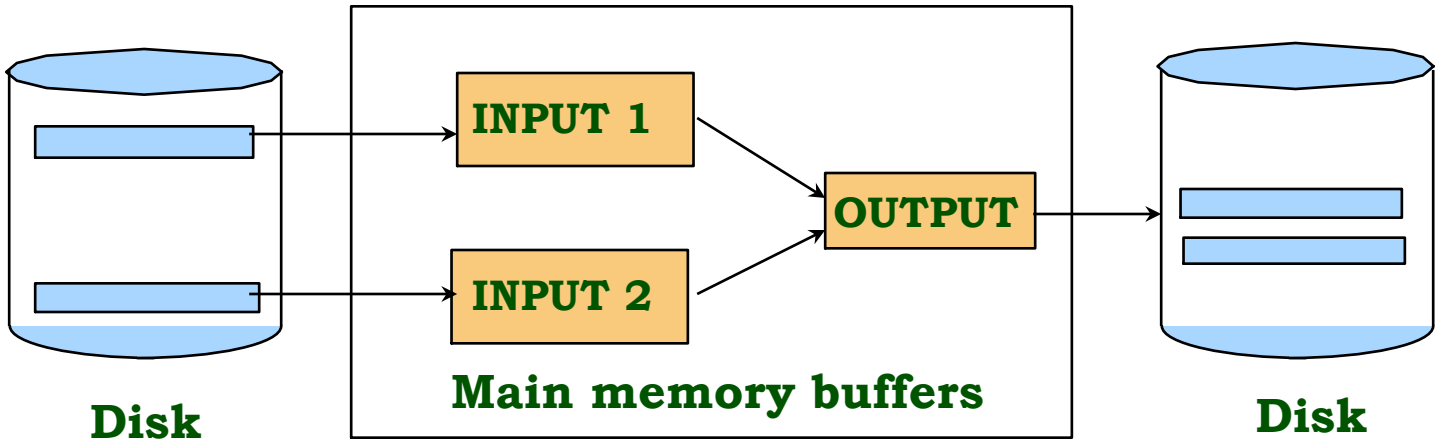
- Important utility in DBMS:
  - *Sort-merge* join algorithm involves sorting.
  - *Eliminate duplicates* in a collection of records (e.g., SELECT DISTINCT)
  - Request data in *sorted order* (e.g., ORDER BY)
    - e.g., find students in decreasing order of *gpa*
  - Sorting is first step in *bulk loading*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

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

2-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

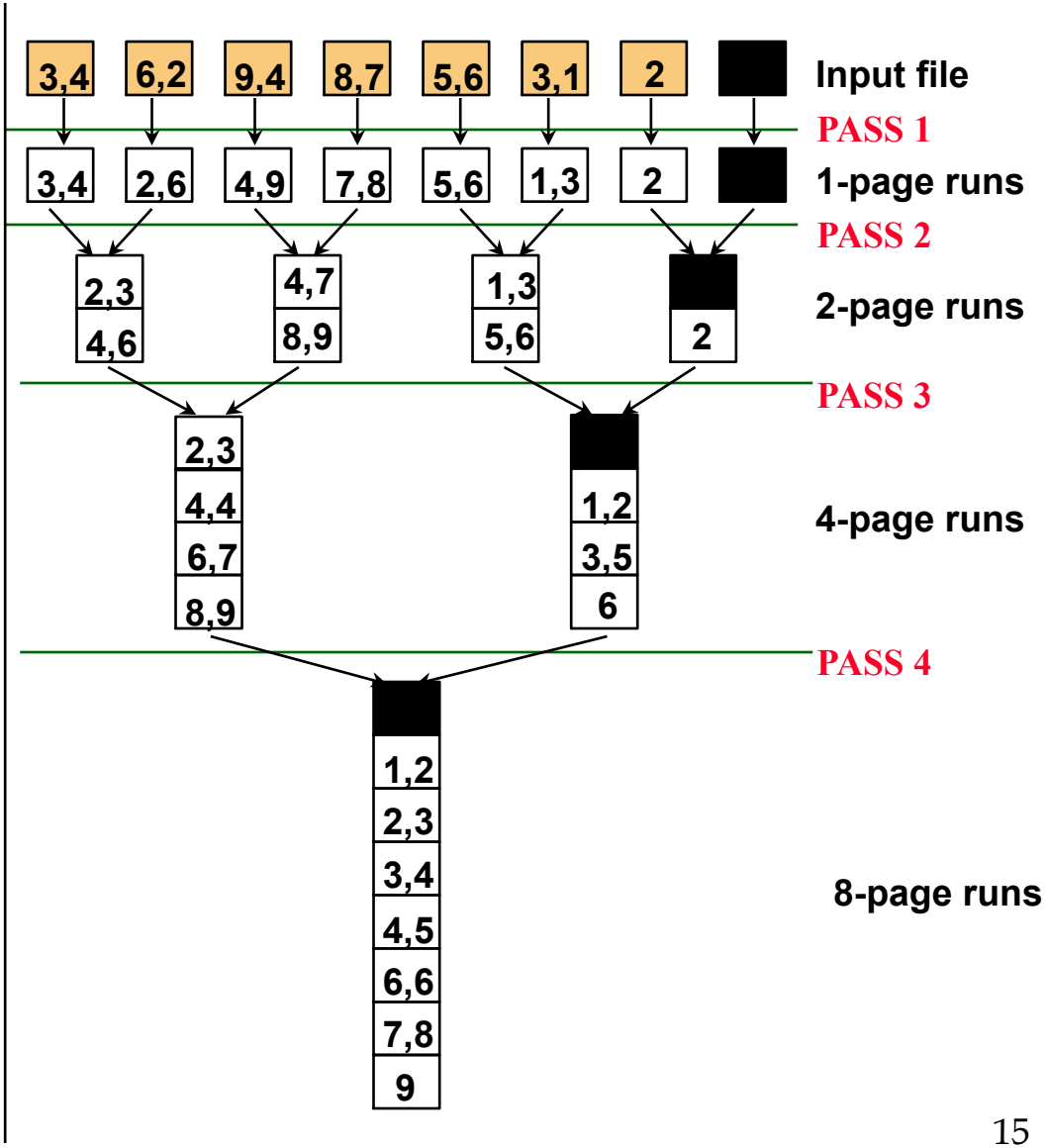
Pass 2: N/2 sorted runs of 2 pages each

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

…

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

$2^P \geq N \rightarrow P \geq \log_2 N$



Two-Way External Merge Sort

- *Divide and conquer*, sort subfiles (runs) and merge

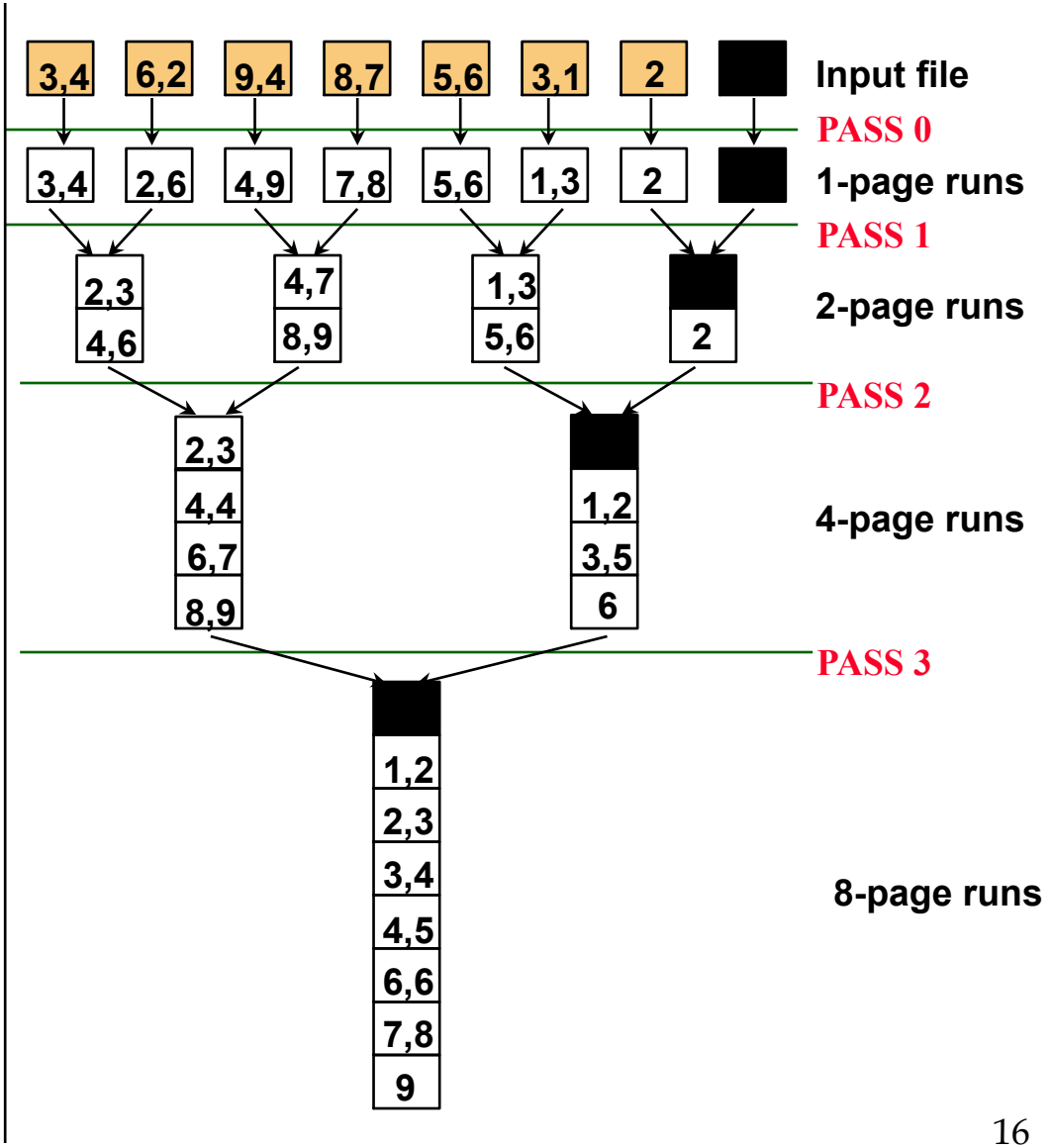
- Each pass, read + write N pages in file → 2N.

- Number of passes is:

$$\lceil \log_2 N \rceil + 1$$

- So total cost is:

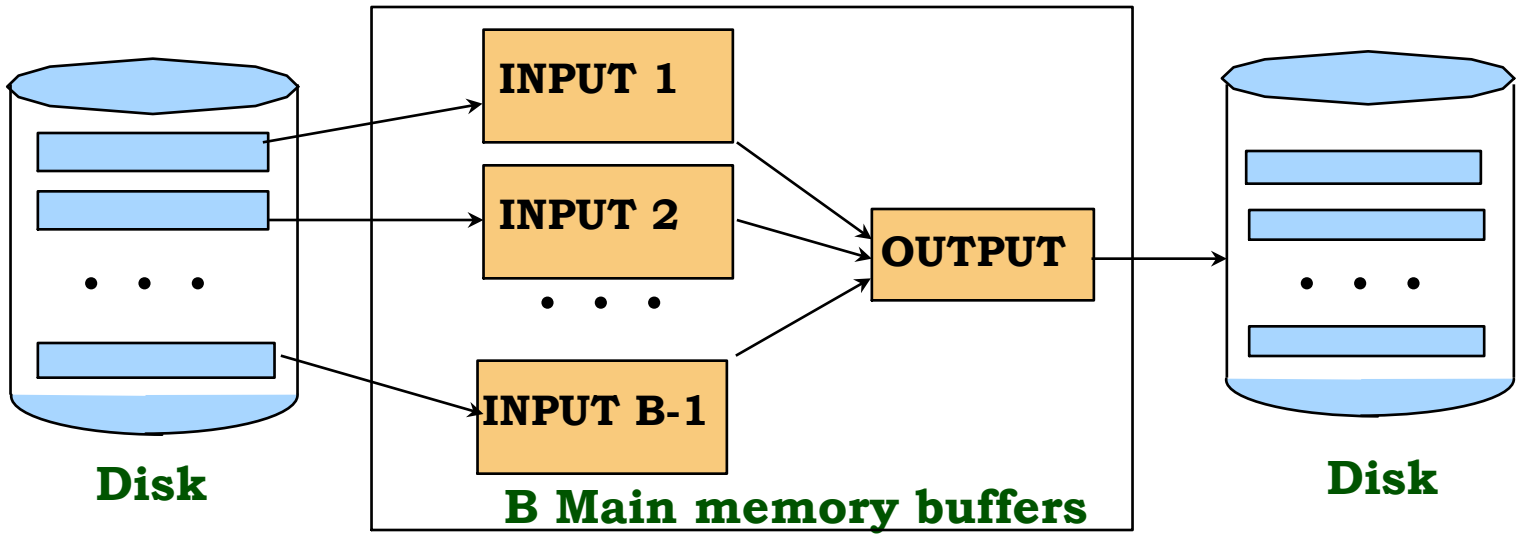
$$2N\left(\lceil \log_2 N \rceil + 1\right)$$



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

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

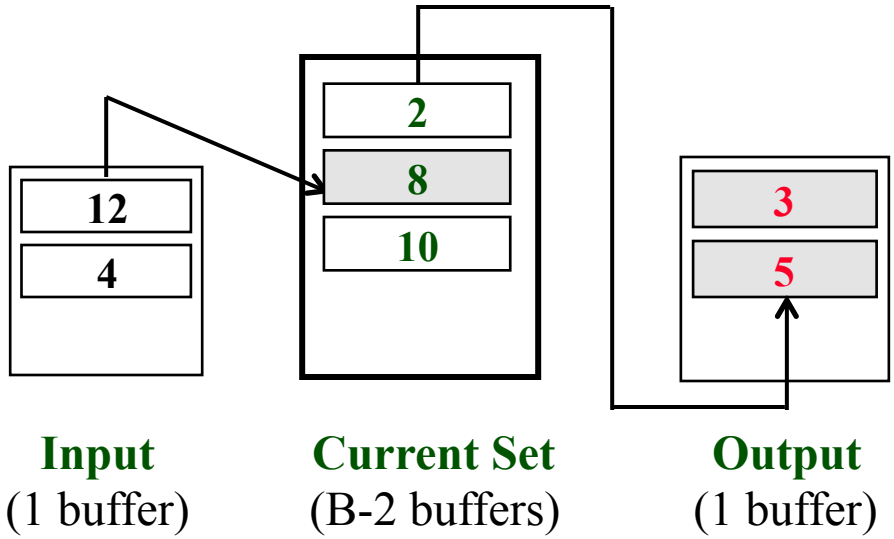
  Cost = $2N * (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$

Number of Passes of External Sort

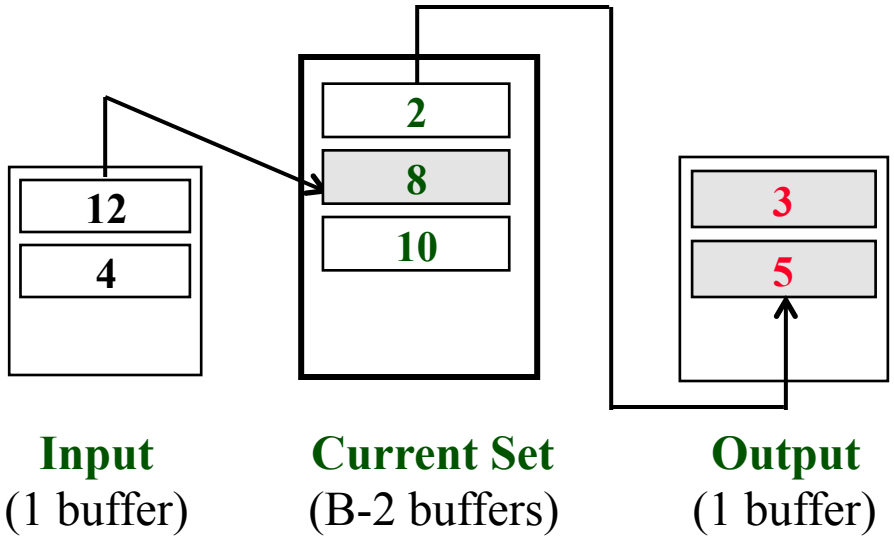
N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

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*

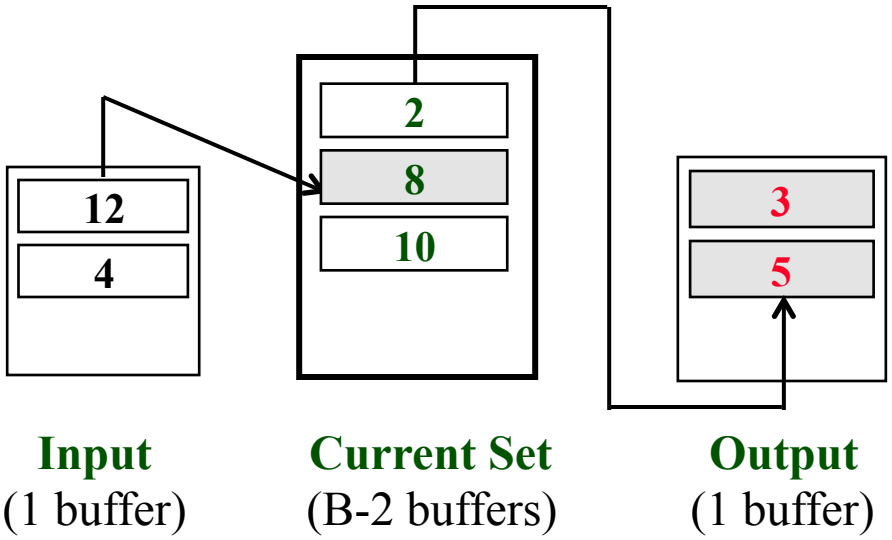


Replacement Sort



- Pick tuple *r* in the current set (CS) s.t. r is the *smallest value* in CS that is ≥ *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



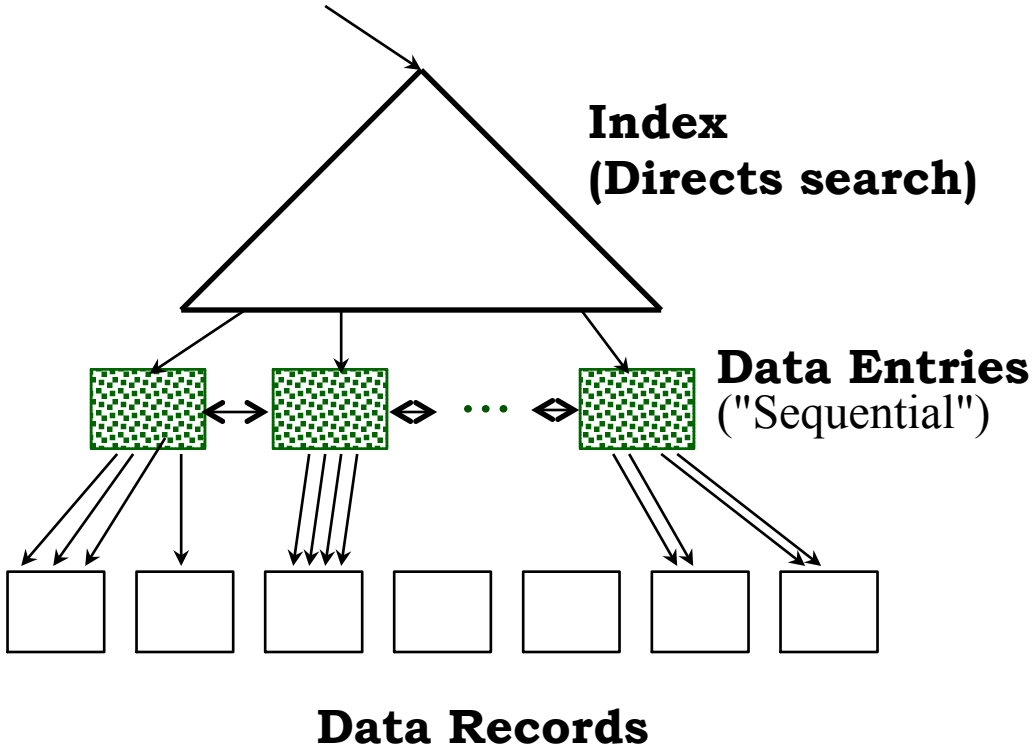
- When used in Pass 1 for sorting, write out sorted runs of size 2B on average.
- Affects calculation of the number of passes accordingly.

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.

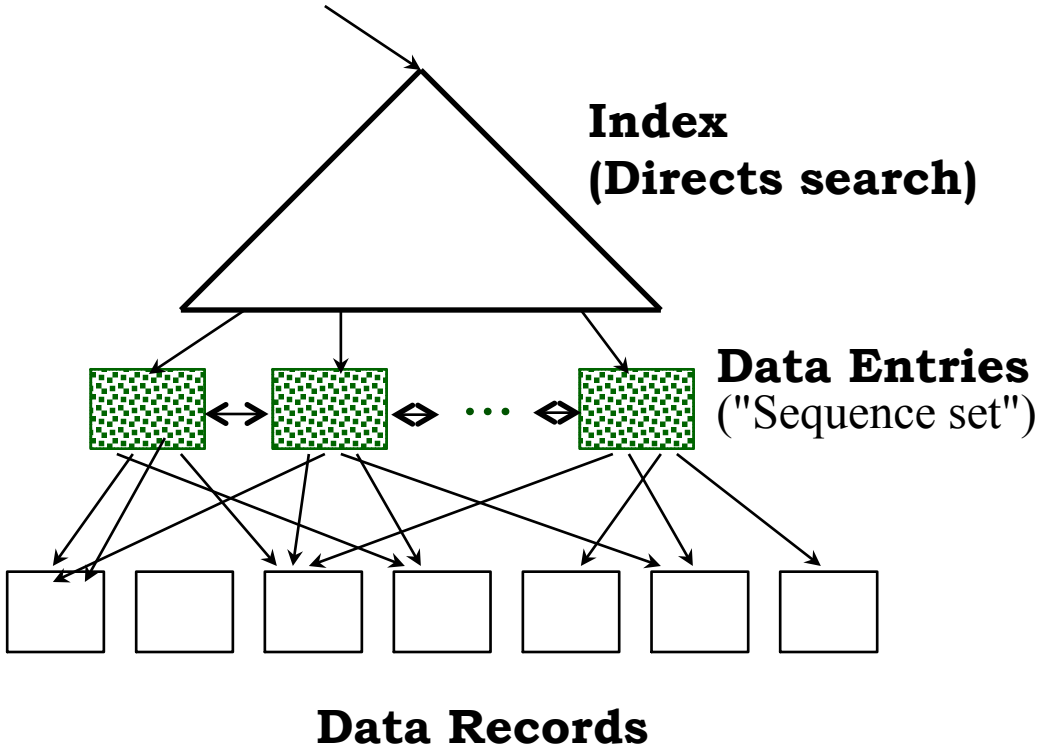


✉ ***Almost always better than external sorting!***

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 (RN) 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
```

- R ⋈ S, natural join. Very common operation!
- Semantics: cross product (×) followed by selection (σ)
  - If R × S is large, R × S followed by a selection is inefficient.
  - Must be carefully optimized.
- *Cost metric*: # of I/Os. Ignore output cost in analysis.
  - R: M pages, p_R tuples per page
  - S: N pages, p_S tuples per page.

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

Page-Oriented Nested Loops Join

- A baseline approach:

```
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
```

- Cost: M + M * N = 1000 + 1000*500 = 501,000 I/Os.
  - 2M random I/Os; others are sequential I/Os.

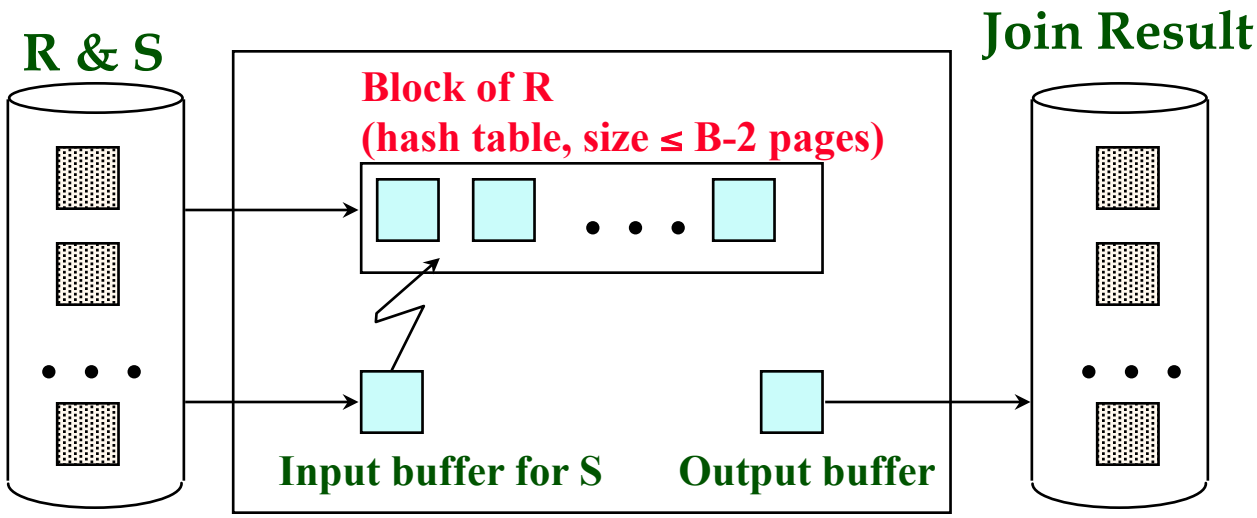
- How many buffers do we need?

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 (Contd.)

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.
  - Which relation should be the outer for smaller cost?

Index Nested Loops Join

- Given an index on the join column of one relation, say S:

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

- Cost: M + (M * p_R * cost of finding matching S tuples)
  1) Cost of search in S index:
     - *Hash index*: about 1.2 I/O to search + extra pages for matches
     - *B+ tree*: 2-4 I/O's to search + extra pages for matches.
  2) Cost of retrieving matching S tuples (assuming Alt. 2 or 3):
     - *Clustered index*: one or a few I/O's (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/O's.

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 ($R \bowtie S$) *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

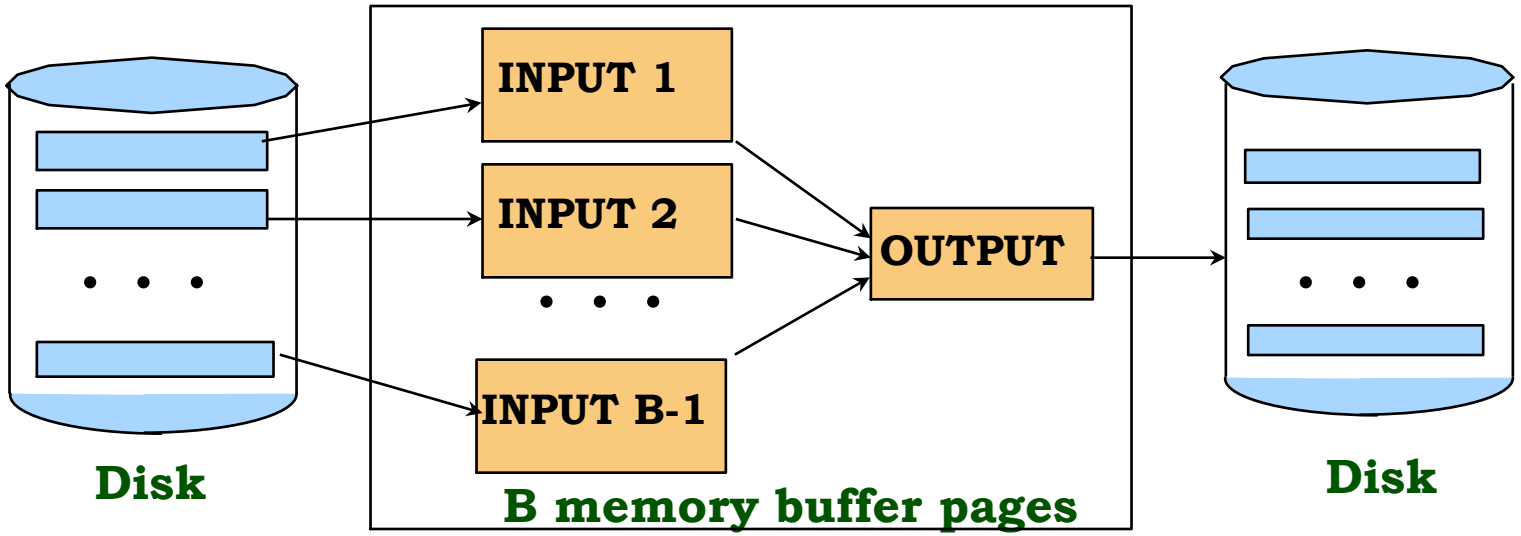
sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

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?
- What is the I/O pattern in the sort-merge join?
- How many buffers are needed in the merge phase?

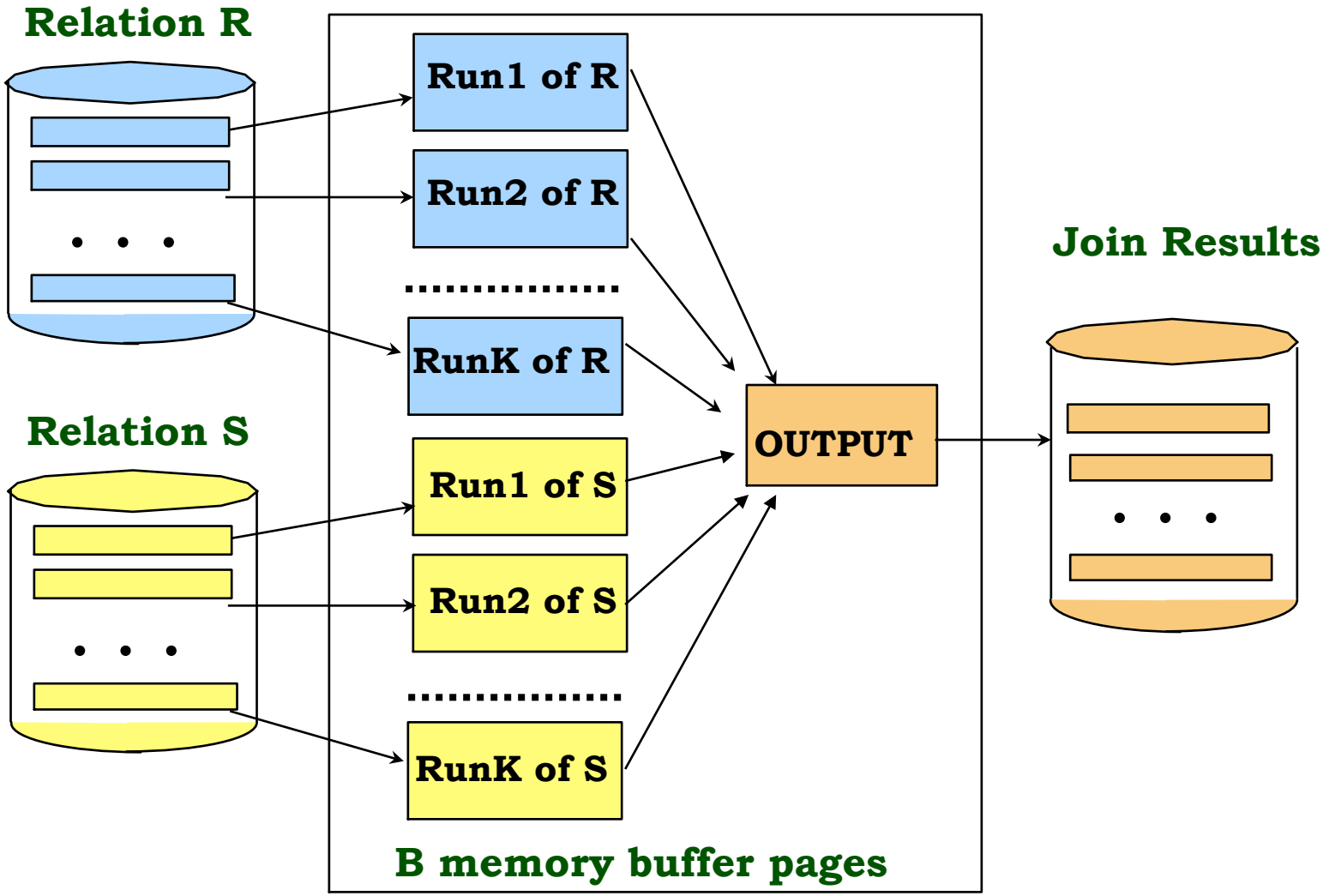
Refinement of Sort-Merge Join

- Is there a guaranteed **linear-time (2 pass)** 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!

Merging in Two-Pass Sort-Merge



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 foreign key-primary key join…

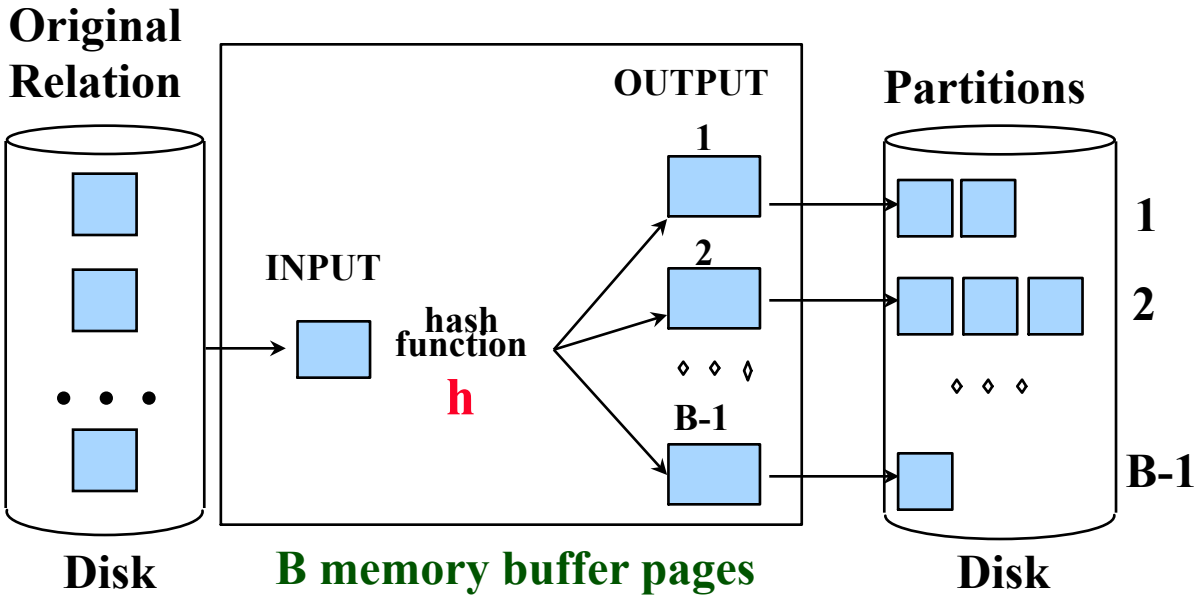
Memory Requirement and Cost

- 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 ≤ U/2B.
  - *Merging* pass holds sorted runs of both relations and an output buffer. So,
    2*(U/2B) + 1 <= B → $\mathbf{B > \sqrt{U}}$

- Cost: read & write each relation in sorting pass
  + read each relation in merging pass
  (+ writing result tuples, ignore here) = 3 (M+N) !

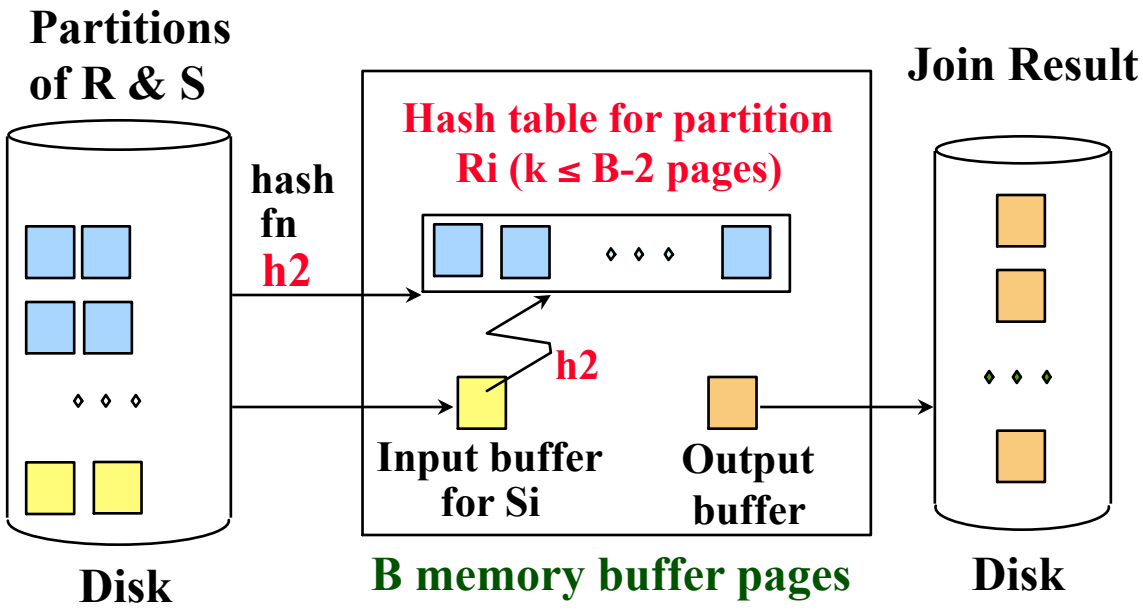
Hash-Join for Equi-Join

- **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 ≤ B-1,
  Probing: to fit each Ri in memory, size of partition ≤ 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…
- What is the I/O pattern in the hash join?

Cost of Hash-Join

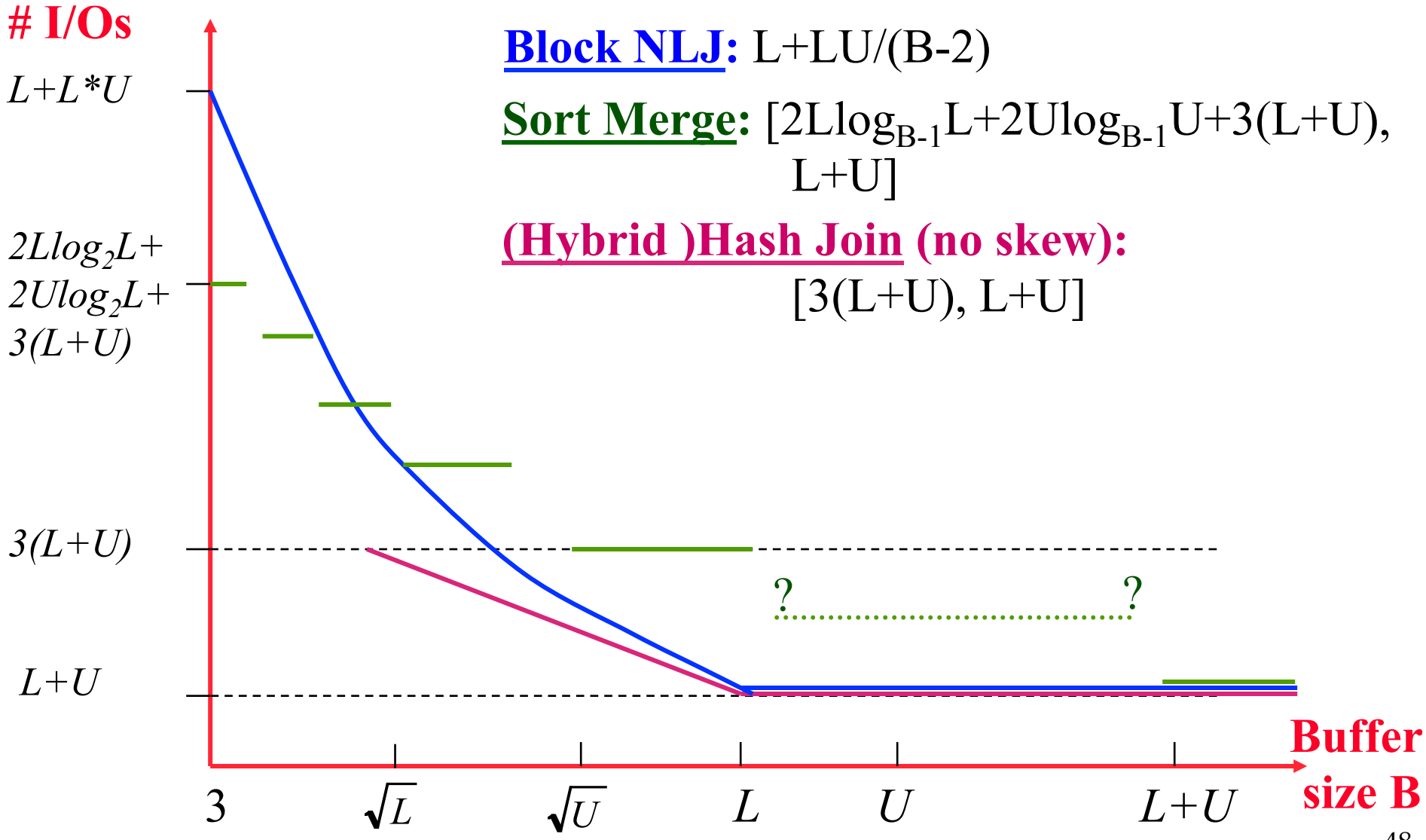
- Partitioning: reads+writes both relns; 2(M+N).

  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.

Rough Comparisons of Join Methods



Block NLJ: L+LU/(B-2)

Sort Merge: [2Llog$_{B-1}$L+2Ulog$_{B-1}$U+3(L+U), L+U]

(Hybrid)Hash Join (no skew): [3(L+U), L+U]

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.