Implementation of Relational Operators/Estimated Cost

1. Select
2. Join
3. Project

Equality Joins With One Join Column

SSN	GPA
123	4.0
234	3.9
456	

R M

SSN	Addr
123	... ames, ia
888	
234	...

S N

SSN	GPA	Addr
123	4.0	... ames, ia
234	3.9	...

R ⋈ S

outer relation

inner relation

```
For each tuple r in R do
{
    for each tuple s in S do
    {
        if r.A == s.B  then add <r, s> to result
    }
}
```

1. CPU Cost? M * N
2. Is R⋈ S the same as S⋈ R?

Simple Nested Loops Join $R \bowtie S$

- Need three pages
  - 1 page for R, 1 page for S, 1 page for output

Algorithm

- Scan the outer relation R
- For each tuple r in R, scan the entire inner relation S

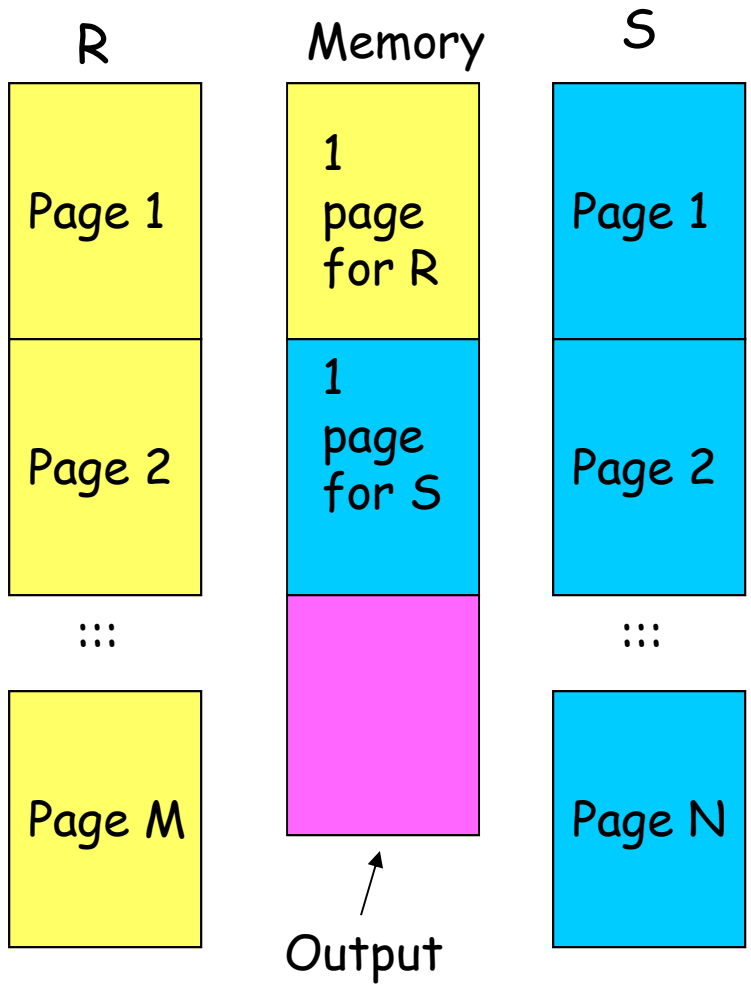
Cost

R has M pages with P_R tuples per page.
S has N pages with P_S tuples per page.

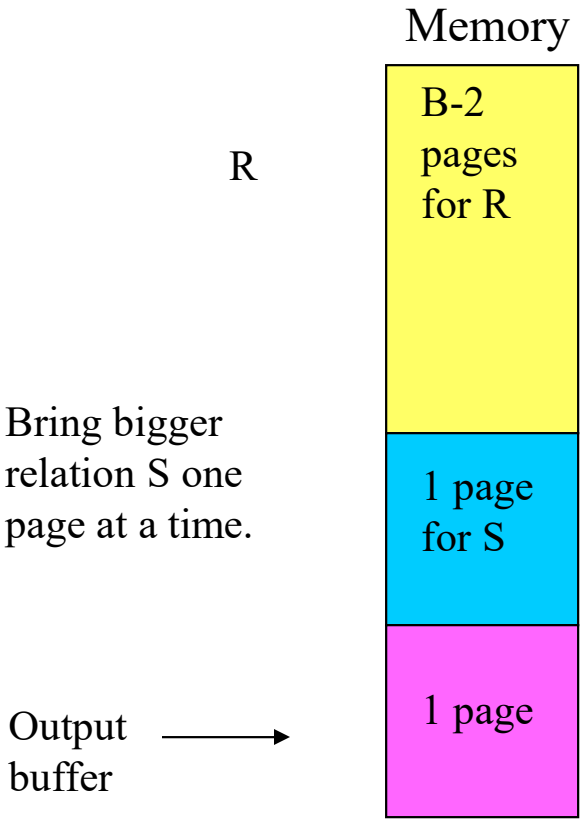
Nested loop join cost = $M+M*P_R*N$.

Cost to scan R.

NOTE: Ignore CPU cost and cost for writing result to disk



Block Nested Loops Join



For each r_i of B-2 pages of R do
  For each s_j of s in S do
    if $r_i.a == s_j.a$ then
  add $<r_i, s_j>$ to result

$$\text{Cost}=M+\left\lceil \frac{M}{B-2} \right\rceil * N$$

Number of blocks of R for each scan of the whole S

B: Available memory in pages.

Cost=M+N → Optimal if one of the relation can fit in the memory (M=B-2).

Indexed Nested Loops Join

```
for each tuple r in R do
        for each tuple s in S do
                if r_i.A == s_j.B  then add <r, s> to result
```

Use index on the joining attribute of S.

Cost=M+M*P_R*(Cost of retrieving a matching tuple in S).

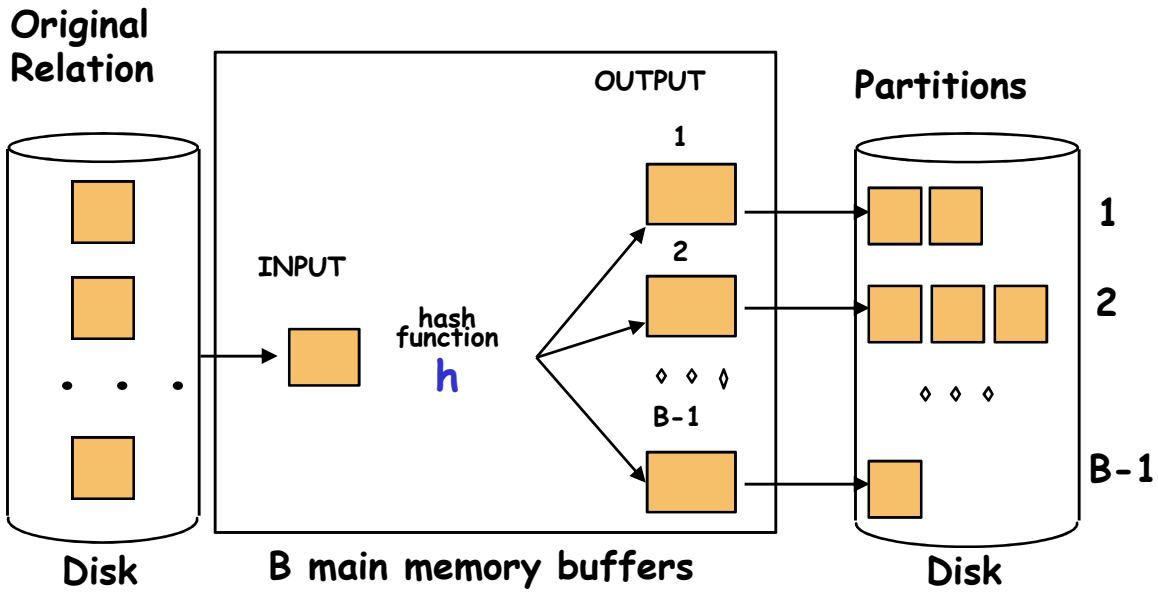
cost to scan R

Depend on the type of index and the number of matching tuples.

Grace Hash-Join

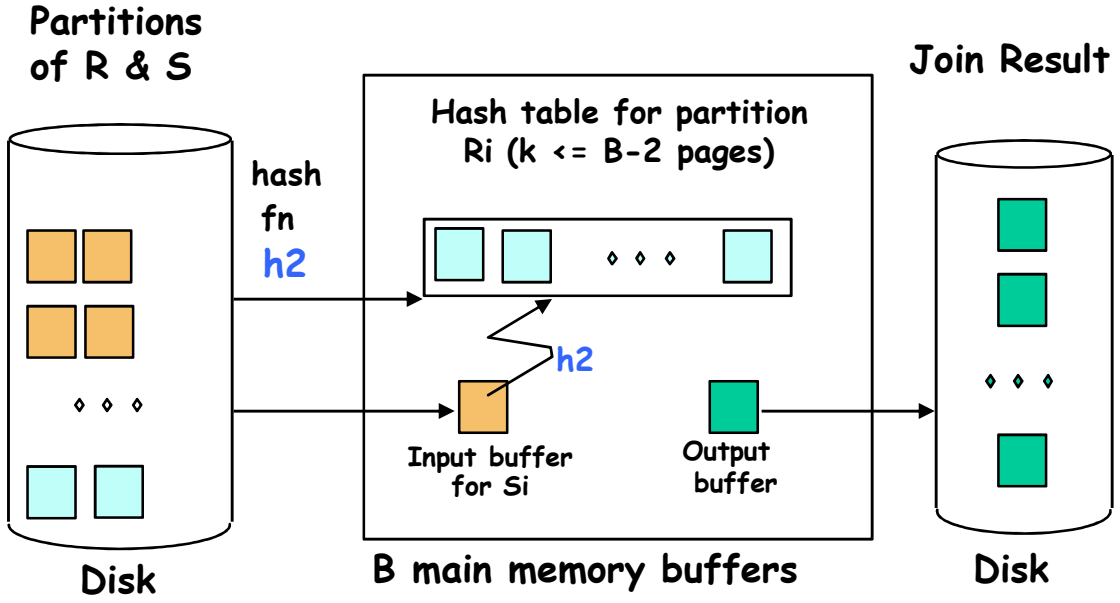
Partition Phase

- Partition both relations using hash function **h**
- R tuples in partition i will only match S tuples in partition i.



Join (probing) Phase

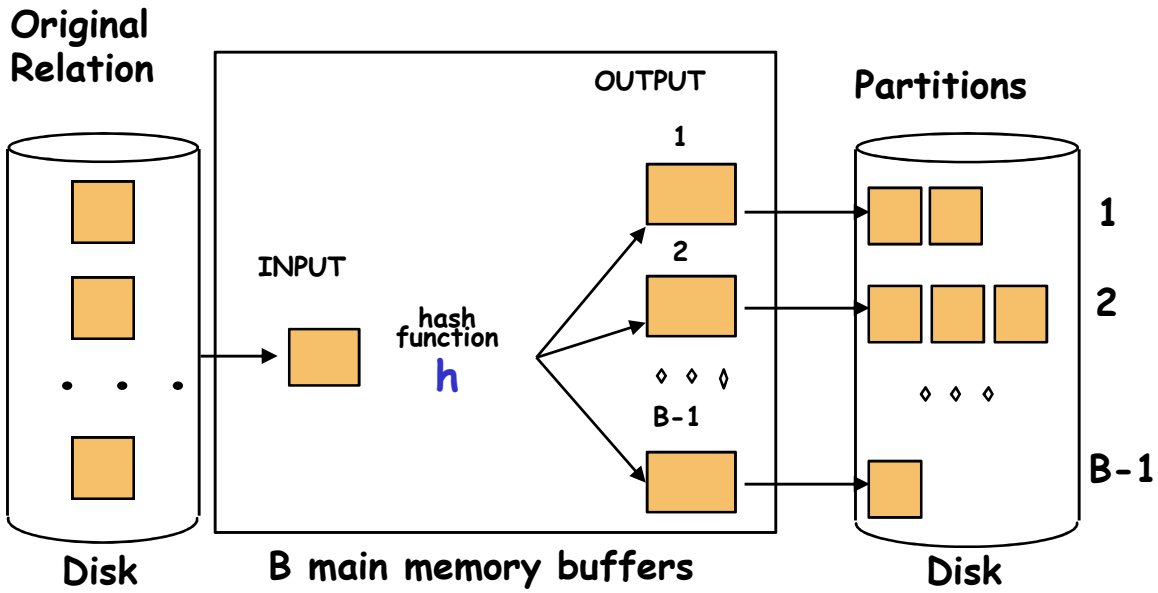
- Read in a partition of R, hash it using **h2 (!=h)**.
- Scan matching partition of S to search for matching tuples



Grace Hash-Join

Partition

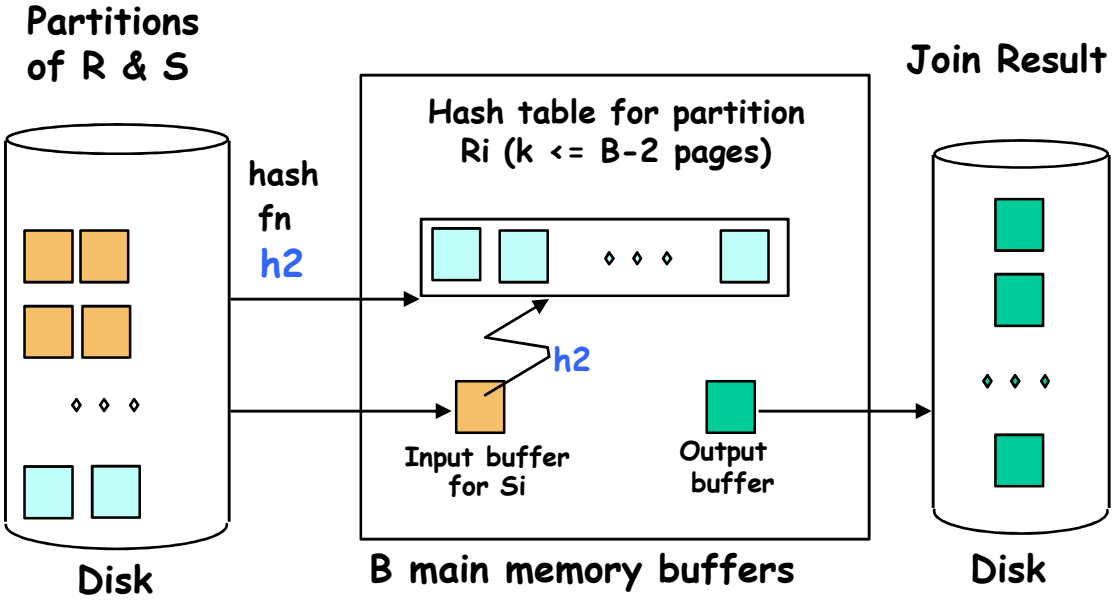
- Partition R into R_1, R_2, ..., R_{B-1}, with h
- Partition S into S1, S_2, ..., S_{B-1}, with h



Join (probing) Phase

Join R_i with S_i

- Partition Ri into $R_{[i,1]}$, $R_{[i,2]}$, ..., $R_{[i,B-2]}$, with h2
- Load each page in Si
- For each record in the page
  - Get its hash value v with h2
  - Match it with $R_{[i,v]}$



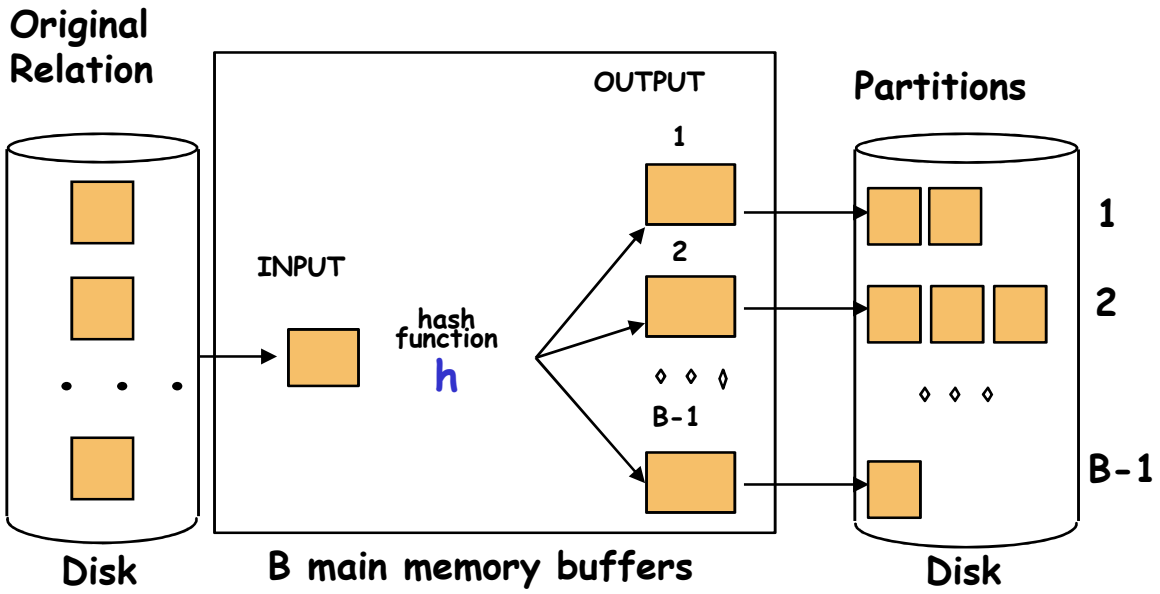
Cost of Grace Join

- Assumption:
  - Each partition fits in the B-2 pages
  - I/O cost for a read and a write is the same
  - Ignore the cost of writing the join results
- Disk I/O Cost
  - Partitioning Phase:
    - I/O Cost: 2*M+2*N
  - Probing Phase
    - I/O Cost: M+N
  - Total Cost 3*(M+N)
- Which one to use, block-nested loop or Grace join?
  - Block-nested loop : $\text{Cost}=M+\left\lceil \frac{M}{B-2} \right\rceil * N$

Memory Requirement for Hash Join

- Ideally, each partition fits in memory
  - To increase this chance, we need to minimize partition size, which means to maximize the number of partitions
- Questions
  - What limits the number of partitions?
  - What is the minimum memory requirement?
    - Partition Phase
    - Probing phase

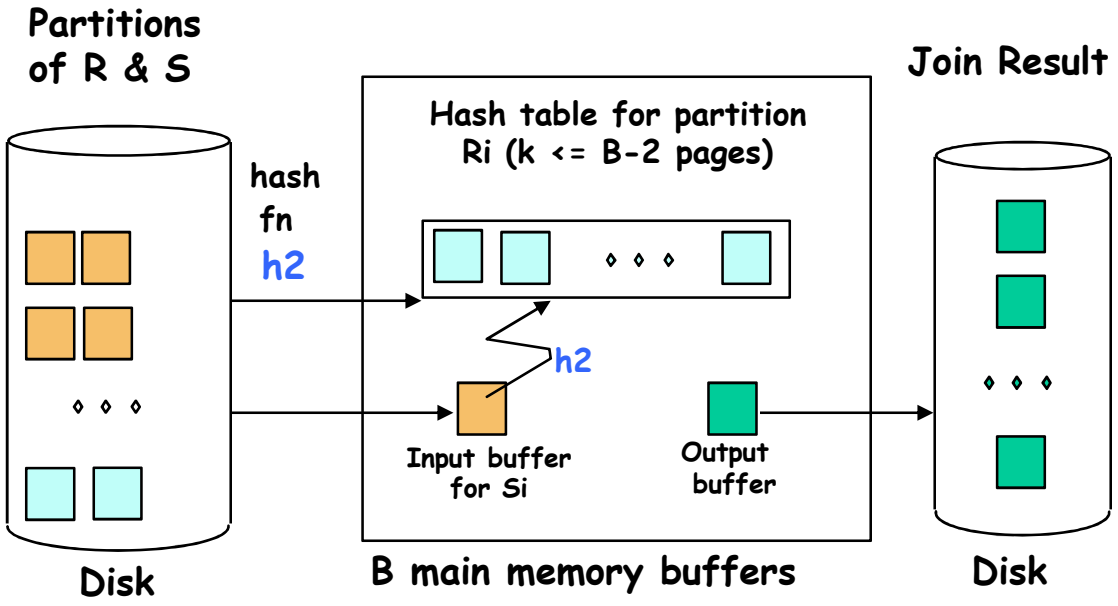
- Partition Phase
  - To partition R into K partitions, we need at least K output buffer and one input buffer
  - Given B buffer pages, the maximum number of partition is B-1
  - Assume a uniform distribution, the size of each R partition is equal to M/(B-1)



- Probing phase
  - # pages for the in-memory hash table built during the probing phase is equal to f*M/(B-1)
    - *f: fudge factor* that captures the increase in the hash table size from the buffer size
  - Total number of pages

$$B > \frac{f \cdot M}{B-1} + 2 \longrightarrow B > \sqrt{f \cdot M}$$

    - since one page for input buffer for S and another page for output buffer



If memory is not enough to store a smaller partition

- Divide a partition of R into sub-partitions using another hash function h3
- Divide a partition of S into sub-partitions using another hash function h3
- Sub-partition *j* of partition *i* in R only matches sub-partition *j* of partition *i* in S

Exercises

- Consider two relations
  - Sailors: 1000 pages, 100 records/page
  - Reserves: 500 pages, 80 records/page
- Estimate the cost of join the two relations with Sailors being the outter relations, with these approaches (ignoring the cost of writing the results)
  - Simple nested loop join
  - Block nested loop join (assuming the total memory is 102 block)
  - Index nested loop join (assuming the inner relation is indexed by a sparse and clustered B^+-tree, and the cost of searching a record takes 4 pages)
  - Grace hash join (assuming the total memory is 102 blocks)
- Same above but with Reserves being the outer relation

Answers

- Consider two relations
  - Sailors: 1000 pages, 100 records/page
  - Reserves: 500 pages, 80 records/page

Simple nested loop join

- Three pages: 1 page for Sailors, 1 page for Reserves, 1 page for output
- Load Sailors page by page
  - The cost is 1000 pages
- For record in the page, scan the whole Reserves
  - The cost is 500 pages
- So the total cost is 1,000 + 1,000 * 100 * 500 pages

Answers

- Consider two relations
  - Sailors: 1000 pages, 100 records/page
  - Reserves: 500 pages, 80 records/page

Block nested loop join

- 100 pages of memory for Sailors, 1 page for Reserves, and 1 page for output
- Each time load 100 pages of Sailors into main memory
- For each load, scan the whole Reserves
- So the total cost is 1,000 + 10* 500 = 6,000 pages

Answers

- Consider two relations
  - Sailors: 1000 pages, 100 records/page
  - Reserves: 500 pages, 80 records/page

Index nested loop join

- Totally the outter relation has 1,000*100 records
- For each record, it needs to load 4 pages to check the matching records
- So the total cost is 1,000 + 1,000*100*4 pages

Answers

- Consider two relations
  - Sailors: 1000 pages, 100 records/page
  - Reserves: 500 pages, 80 records/page

Grace hash join

- Partitioning
  - Sailors: 1,000 reads and 1,000 writes
  - Reserves: 500 reads and 500 writes
- Probing
  - Sailors: 1000 reads
  - Reserves: 500 reads
- Total: (1000 + 500) * 3 pages

Sort-Merge Join ($R \bowtie_{i=j} S$)

Sailors

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

Reserves

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

- I/O Cost for Sort-Merge Join
  - Cost of sorting: TBD
    - O(|R| *log* |R|)+O(|S| *log* |S|) ??
  - Cost of merging: M+N
    - Could be up to O(M+N) if the inner-relation has to be scanned multiple times (very unlikely)

Sort-Merge Join ($R \bowtie_{i=j} S$)

- Sort R and S on the join attribute, then scan them to do a ``merge'' (on join column), and output result tuples
  - Advance scan of R until current tuple R.i >= current tuple S.j, then advance scan of S until current S.j >= current R.i; do this until current R.i = current S.j.
  - At this point, all R tuples with same value in R.i (*current R partition*) and all S tuples with same value in S.j (*current S partiton*) *match*;
    - output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.

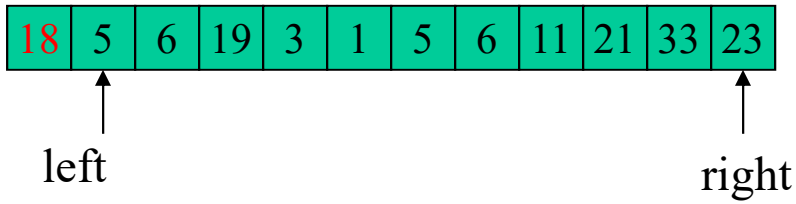
Sort-merge join is attractive if one relation is already sorted on the join attribute or has a clustered index on the join attribute

In-memory Sort: Quick Sort

1. pick one element in the array, which will be the *pivot*.
2. make one pass through the array, called a *partition* step, re-arranging the entries so that:
    a) the pivot is in its proper place.
    b) entries to the left of the pivot are smaller than the pivot
    c) entries to its right are larger than the pivot

    Detailed Steps:
    1. Starting from left, find the item that is larger than the pivot,
    2. Starting from right, find the item that is smaller than the pivot
    3. Switch left and right
3. recursively apply quicksort to the part of the array that is to the left of the pivot, and to the part on its right.

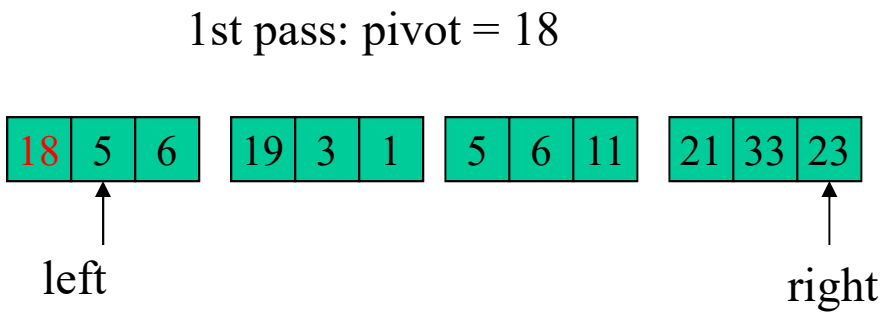
1st pass: pivot = 18



O(n log n) on average, worst case is $O(n^2)$

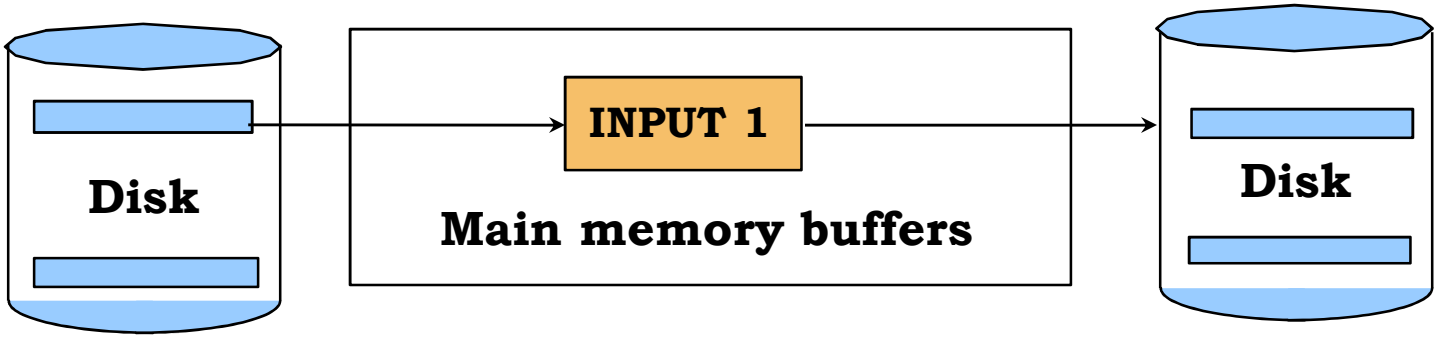
Sorting large amount of data

- Sort 1Gb of data with 1Mb of RAM?
- Why not just using virtual memory?

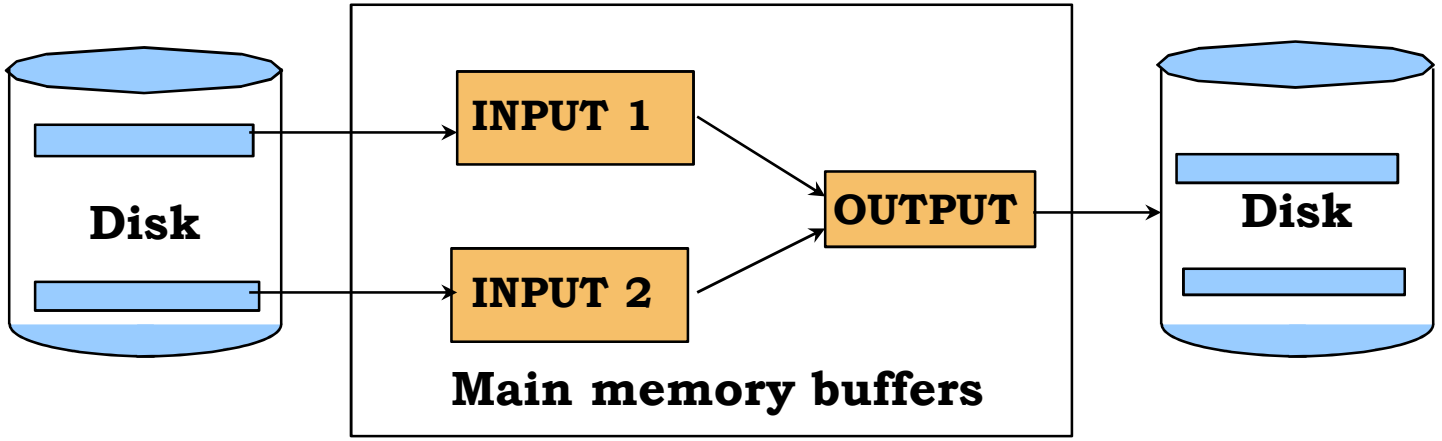


2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it
  - only one buffer page is used



- Pass 2, 3, …, etc.:
  - three buffer pages used.



Two-Way External Merge Sort

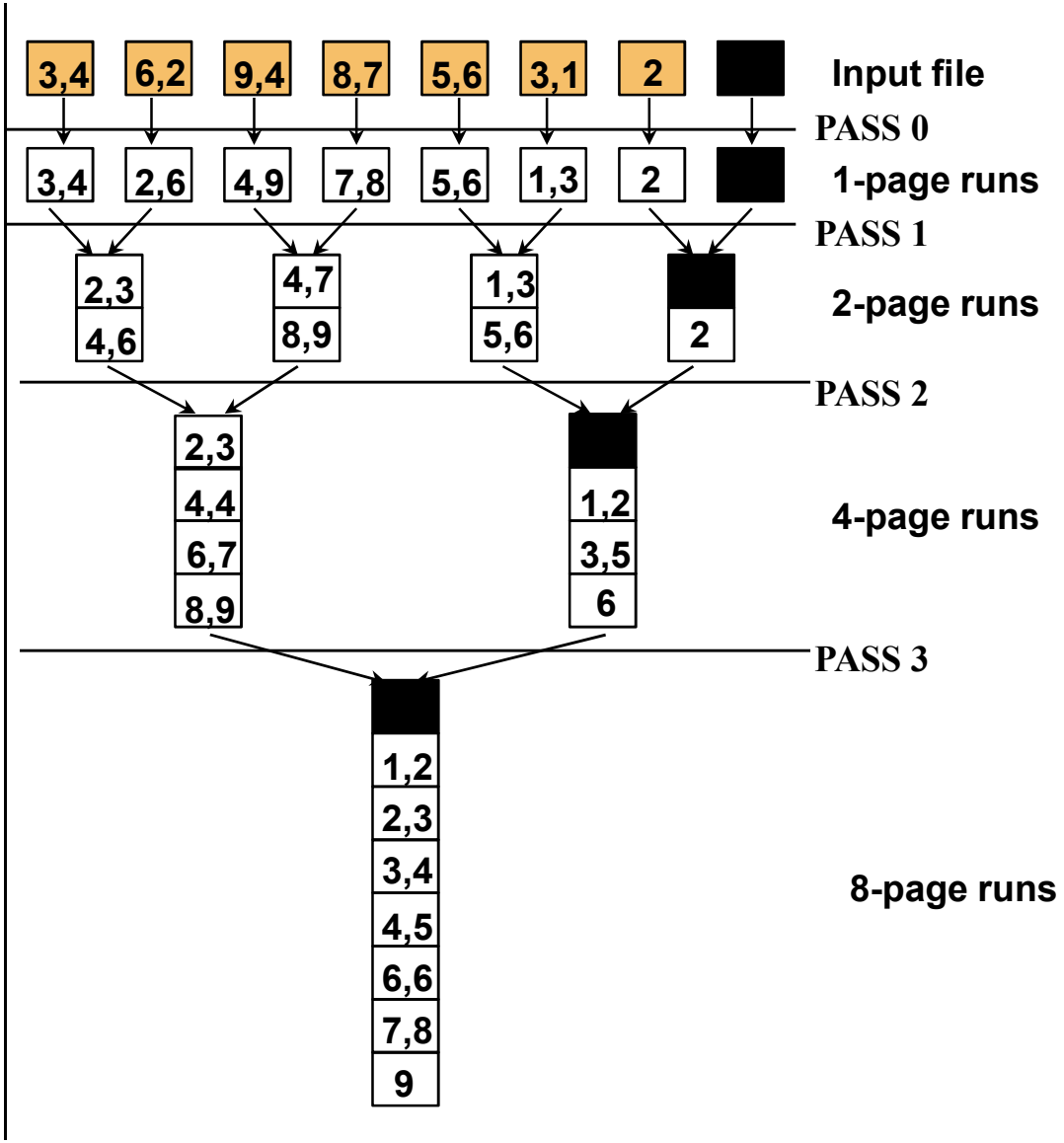
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

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

- So total cost is:

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

- *Idea:* ***Divide and conquer:*** sort subfiles and merge

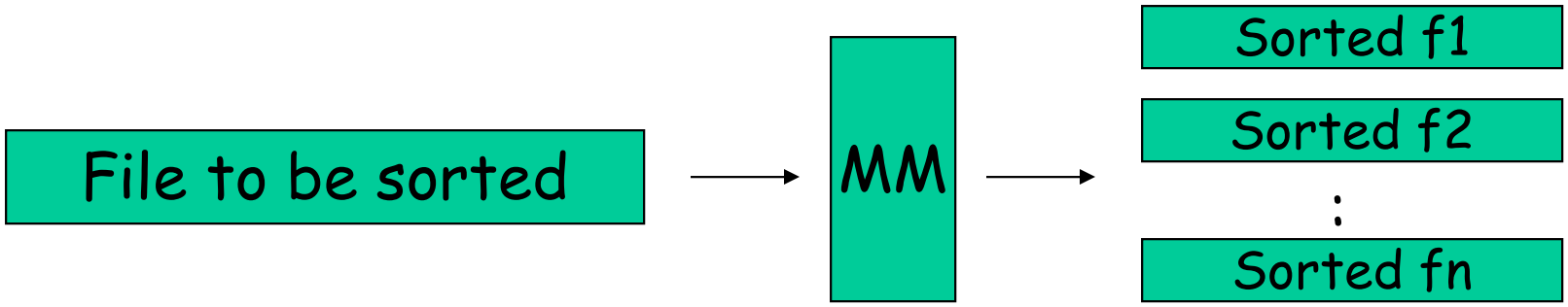


Two-Phase Multi-Way Merge-Sort

More than 3 buffer pages. How can we utilize them?

Phase 1:

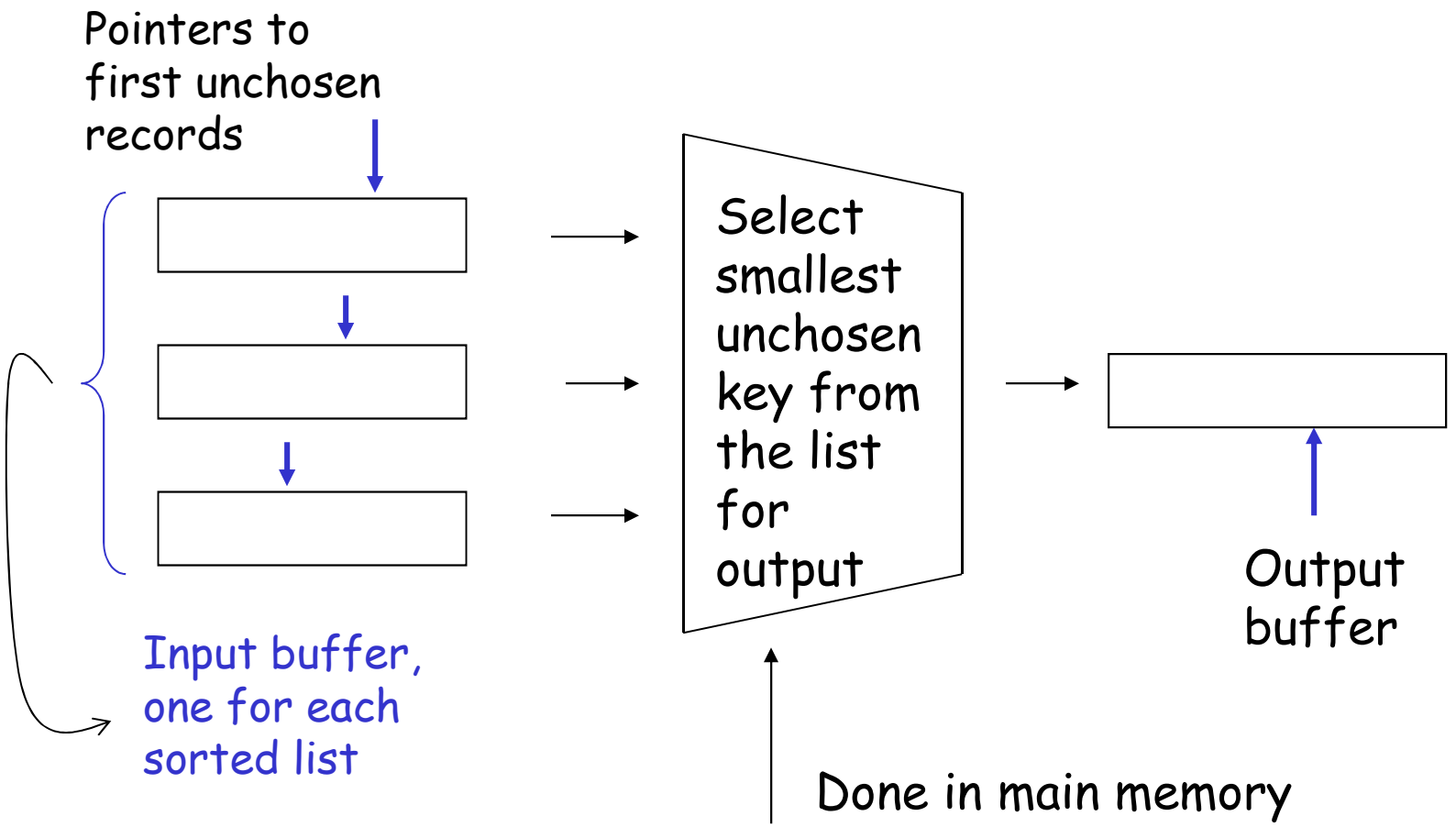
1. Fill all available main memory with blocks from the original relation to be sorted.
2. Sort the records in main memory using main memory sorting techniques.
3. Write the sorted records from main memory onto new blocks of secondary memory, forming one sorted sublist. (There may be any number of these sorted sublists, which we merge in the next phase).



Phase 2: Multiway Merge-Sort

Merge all the sorted sublists into a single sorted list.

- Partition MM into n blocks
- Load Fi to block i



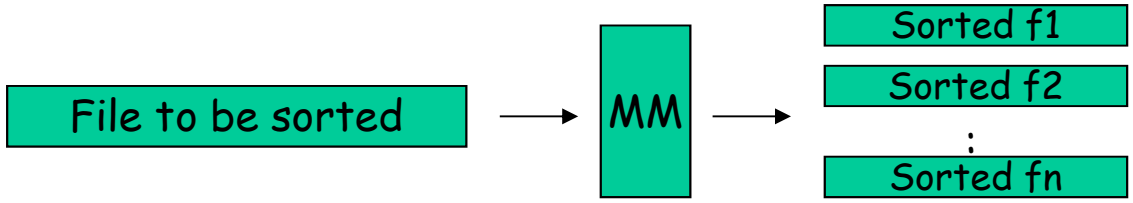
Phase 2: Multiway Merge-Sort

- Find the smallest key among the first remaining elements of all the lists. (This comparison is done in main memory and a linear search is sufficient. Better technique can be used.)
- Move the smallest element to the first available position of the output block.
    - If the output block is full, write it to disk and reinitialize the same buffer in main memory to hold the next output block.
- If the block from which the input smallest element was taken is now exhausted, read the next block from the same sorted sublist into the same buffer.
- If no block remains, leave its buffer empty and do not consider elements from that list.

Memory Requirement for Multi-Way Merge-Sort

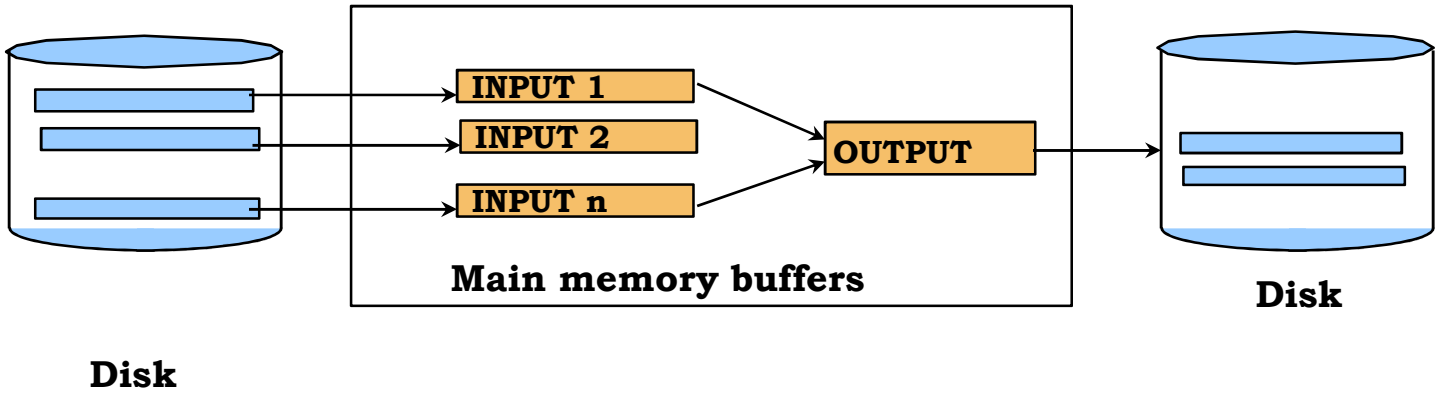
Partitioning Phase:

- Given M pages of main memory and a file of |f| pages of data, we can partition the file into |f|/M small files.



Merge Phase:

- At least M-1 pages of main memory are needed to merge |f|/M sorted lists, so we have M-1>=|f|/M, i.e., M(M-1)>=|f|.



Exercises

- Consider two relations
  - Sailors: 1000 pages, 100 records/page
  - Reserves: 500 pages, 80 records/page
- Estimate the cost of join the two relations with Sailors being the outter relations, with these approaches (ignoring the cost of writing the results)
  - Simple nested loop join
  - Block nested loop join (assuming the total memory is 102 blocks)
  - Index nested loop join (assuming the inner relation is indexed by a sparse and nonclustered B^+-tree, and the cost of searching a record takes 4 pages)
  - Grace hash join (assuming the total memory is 102 blocks)
  - Sort-merge join (assuming two-way sort-merge approach)
- Same above but with Reserves being the outer relation

Exercises

- Consider two relations
  - Sailors: 1000 pages, 100 records/page
  - Reserves: 500 pages, 80 records/page

Answer

- Phase I: SORT
  - Sort Sailors in log(1,000)=10 passes, each of which needs to read and write the whole relation
    - the cost is 10 * 1,000 * 2 = 20,000 pages
  - Sort Reserves in log(500) = 9 passes, each of which needs to read and write the whole relations
    - the cost is 9 * 500 * 2 = 9,000 pages
- Phase II: MERGE
  - Need to load both relations into the main memory
    - the cost is 1,000+500 =1,500 pages (ignoring the cost of writing the results to the disk)
- Total cost = 20,000+9,000+1,500 pages

Query Cost Estimation

1. Select: $\sigma_c(R)$
   - No index
     - unsorted data
     - sorted data
   - Index
     - tree index
     - hash-based index

2. Join: R $\bowtie$ S
   - Simple nested loop
   - Block nested loop
   - Grace Hash
   - Sort-merge