

CS150A Database

Lu Sun

School of Information Science and Technology

ShanghaiTech University

Oct. 13, 2022

Today:

- Sorting and Hashing:
 - Double Buffering
 - Divide & Conquer
 - Parallelize Sorting and Hashing

Readings:

- Database Management Systems (DBMS), Chapters 13.1-13.3, 13.4.2

Why Sort?

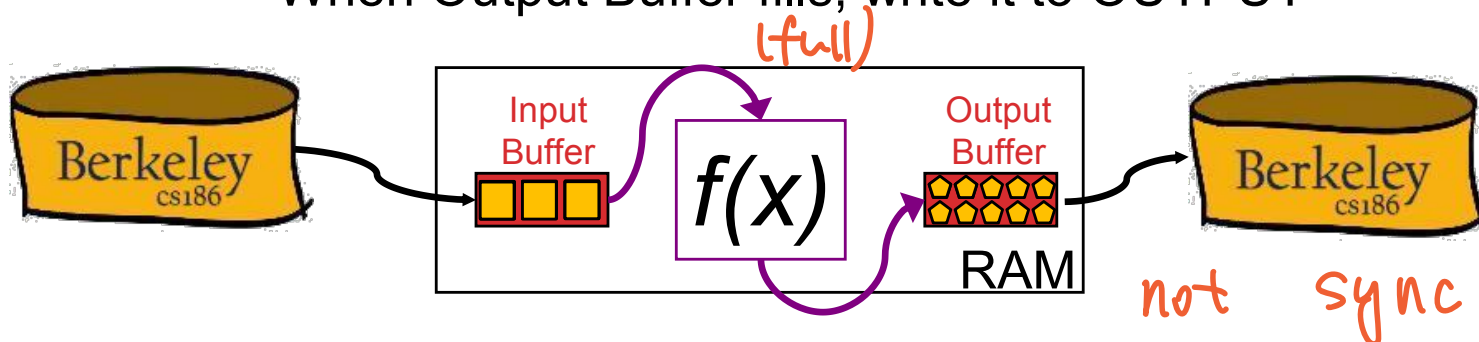
- “Rendezvous”
 - Eliminating duplicates (DISTINCT)
 - Grouping for summarization (GROUP BY)
 - Upcoming sort-merge join algorithm
- Ordering
 - Sometimes, output must be ordered (ORDER BY)
 - e.g., return results ranked in decreasing order of relevance
 - First step in bulk-loading tree indexes
- Problem: sort 100GB of data with 1GB of RAM.
 - why not virtual memory?

Out-of-Core Algorithms

- Two themes
 1. Single-pass streaming data through RAM
 2. Divide (into RAM-sized chunks) and Conquer

Single-pass Streaming

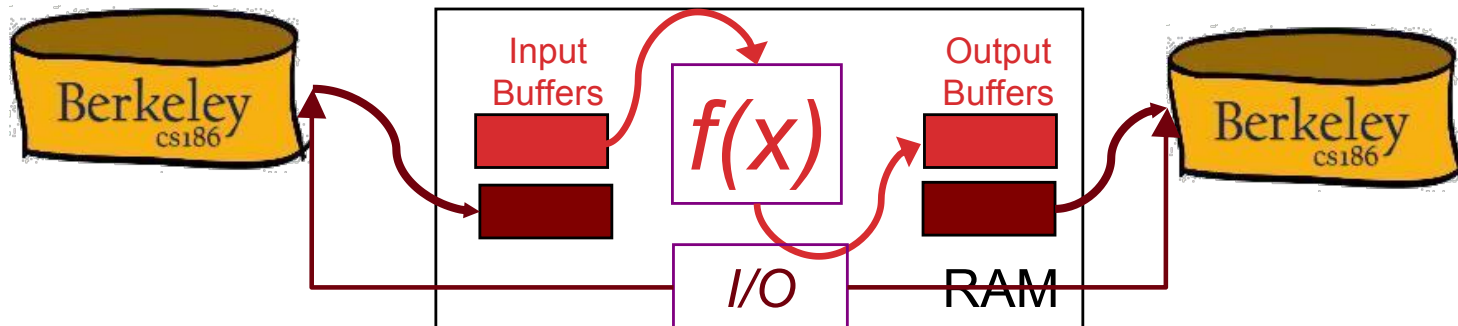
- Simple case: “Map”.
 - Goal: Compute $f(x)$ for each record, write out the result
 - Challenge: minimize RAM, call read/write rarely
- Approach
 - Read a chunk from INPUT to an Input Buffer
 - Write $f(x)$ for each item to an Output Buffer
 - When Input Buffer is consumed, read another chunk
 - When Output Buffer fills, write it to OUTPUT



Better: Double Buffering pt 1

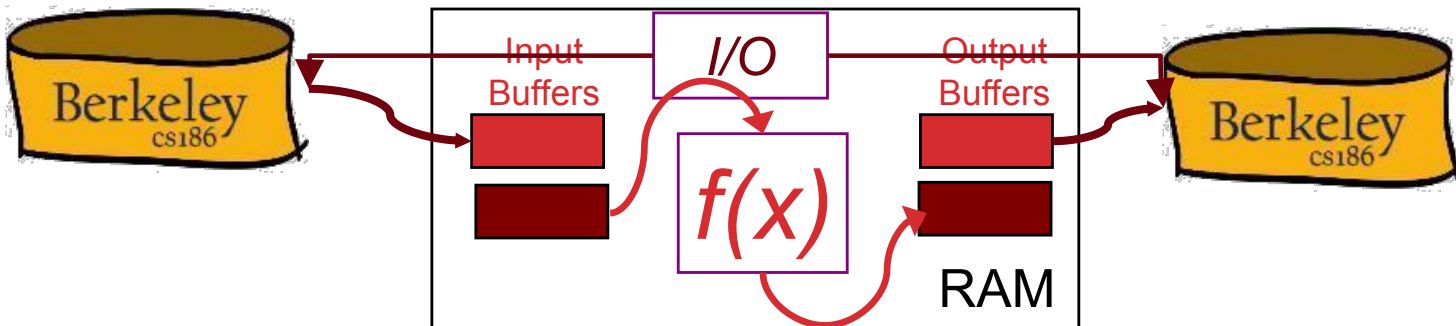
- **Main thread** runs $f(x)$ on one pair I/O bufs
- 2nd **I/O thread** drains/fills unused I/O bufs **in parallel**
 - Why is parallelism available?
 - Theme: I/O handling usually **deserves its own thread**
- Main thread ready for a new buf? Swap!

compute
I/O



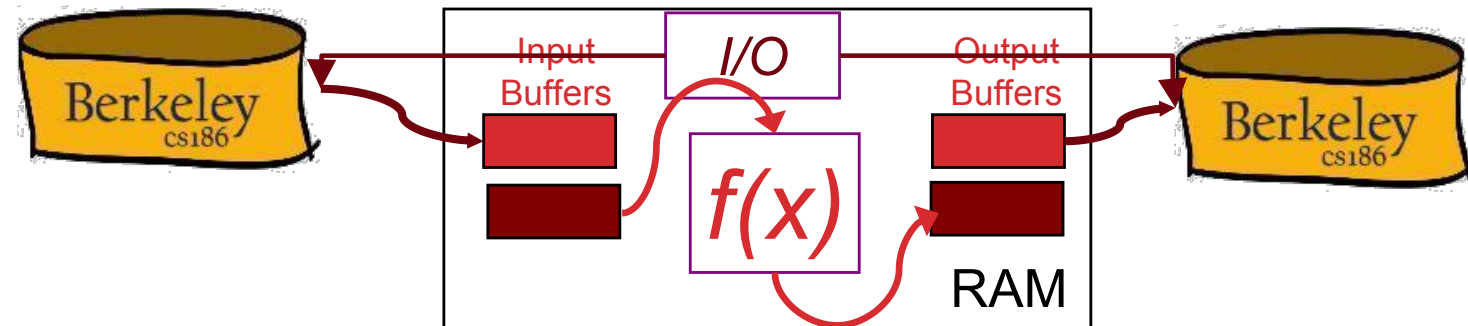
Better: Double Buffering pt 2

- **Main thread** runs $f(x)$ on one pair I/O bufs
- **2nd I/O thread** drains/fills unused I/O bufs in parallel
 - Why is parallelism available?
 - Theme: I/O handling usually deserves its own thread
- Main thread ready for a new buf? **Swap!**



Double Buffering applies to all streams

- Usable in any of the subsequent discussion
 - Assuming you have RAM buffers to spare!
 - But for simplicity we won't bring this up again.



Sorting & Hashing: Formal Specs

Sorting

- Produce an output file F_S
 - with contents R **stored in order by a given sorting criterion**

Hashing

- Produce an output file F_H
 - with contents R , **arranged on disk so that no 2 records that have the same hash value are separated by a record with a different hash value.**
 - I.e. matching records are always “stored consecutively” in F_H .

continuously

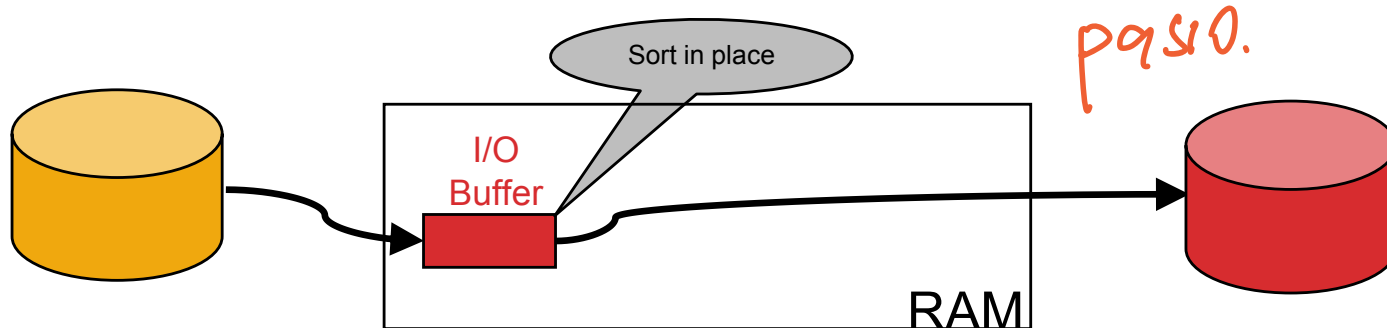
Given:

- A file F :
 - containing a multiset of records R
 - consuming N blocks of storage
- Two “scratch” disks
 - each with $\gg N$ blocks of free storage
- A fixed amount of space in RAM
 - memory capacity equivalent to B blocks of disk

h₁ h₂ h₁
h₁ h₁ h₂ ✓

Sorting: 2-Way (a strawman)

- Pass 0 (conquer a batch):
 - read a page, sort it, write it.
 - only one buffer page is used
 - a repeated “batch job”

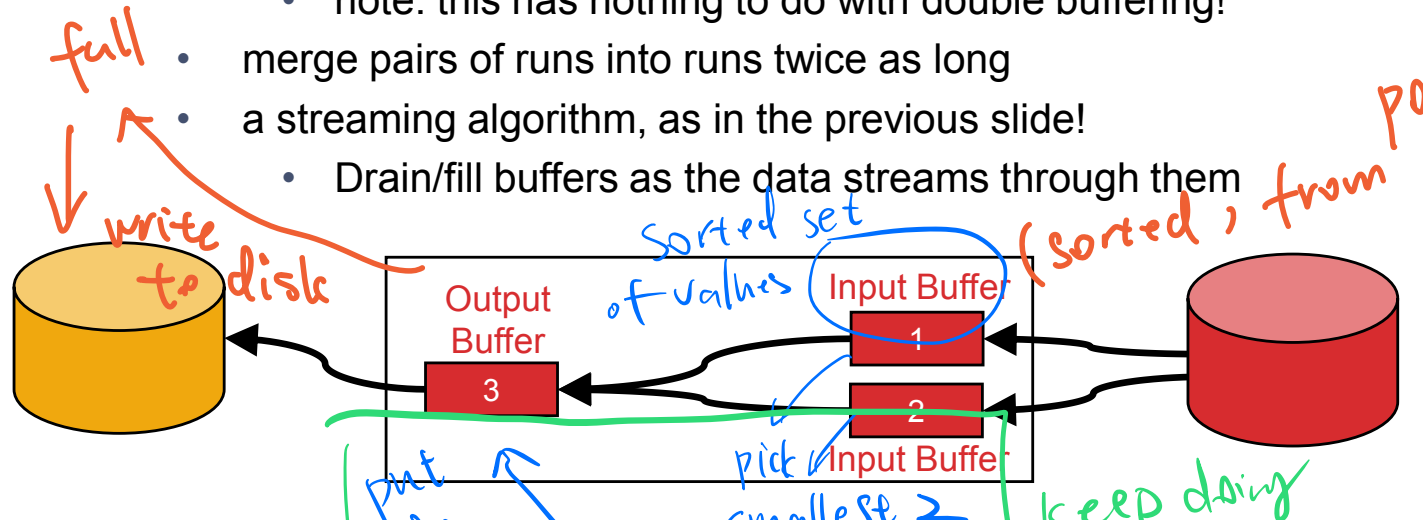


Take 2 sorted blocks.

Sorting: 2-Way (a strawman), cont

merging to 1 sorted run.

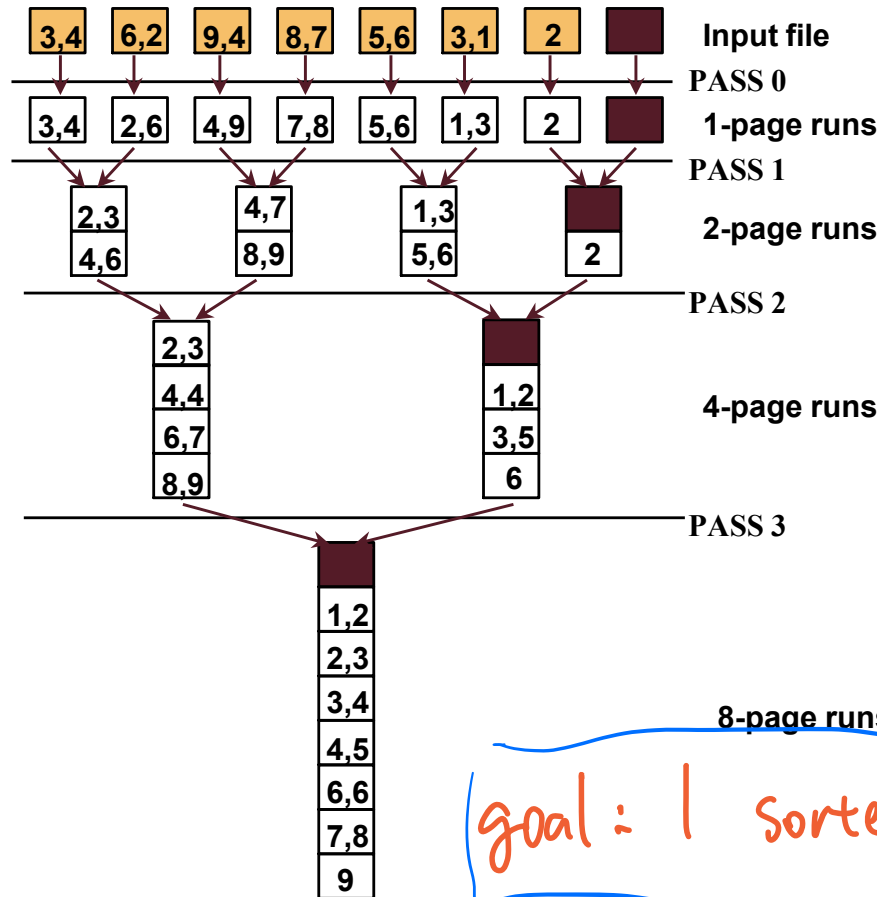
- Pass 0 (conquer a batch):
 - read a page, sort it, write it.
 - only one buffer page is used
 - a repeated “batch job”
- Pass 1, 2, 3, ..., etc. (merge via streaming):
 - requires 3 buffer pages
 - note: this has nothing to do with double buffering!
 - merge pairs of runs into runs twice as long
 - a streaming algorithm, as in the previous slide!
 - Drain/fill buffers as the data streams through them



#

not page
of block
(chunk)

Two-Way External Merge Sort



- Conquer and Merge:
 - sort subfiles and merge

- Each pass we read + write each page in file ($2N$)

- N pages in the file.

- So, the number of passes is: $\lceil \log_2 N \rceil + 1$

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

can only group 2

goal: 1 sorted file. at a time.

N : file consuming N blocks.

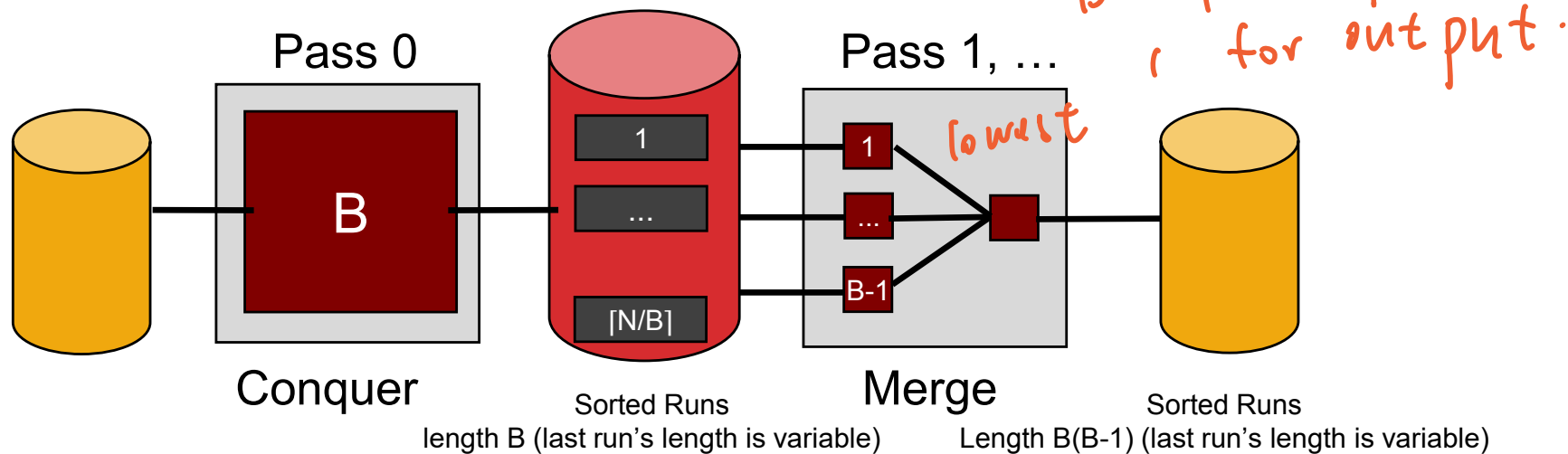
pass 0.

2: read and write: N ; N blocks.

?

General External Merge Sort

- More than 3 buffer pages. How can we utilize them? $B > 3$.
 - Big batches in pass 0, many streams in merge passes
- To sort a file with N pages using B buffer pages:
 - Pass 0: use B buffer pages. Produce N/B sorted runs of B pages each.
 - Pass 1, 2, ..., etc.: merge $B-1$ runs at a time.



Cost of External Merge Sort

- Number of passes: $1 + \log_{B-1} N / B$

Cost = $2N * (\# \text{ of passes})$ *fanout.*

$\frac{N}{B}$: # of batch (Sorted)

- E.g., with 5 buffer pages, to sort 108 page file:
 - Pass 0: $\lceil 108 / 5 \rceil = 22$ sorted runs of 5 pages each
 - last run is only 3 pages
 - Pass 1: $\lceil 22 / 4 \rceil = 6$ sorted runs of 20 pages each
 - last run is only 8 pages $\rightarrow 6 \div 4 = 2$ (batch)
 - Pass 2: 2 sorted runs, 80 pages and 28 pages
 - Pass 3: Sorted file of 108 pages

N, B

often
in form

of (pages)

Formula check: $1 + \lceil \log_4 22 \rceil = 1 + 3 \rightarrow \underline{4 \text{ passes}}$ ✓

of Passes of External Sort

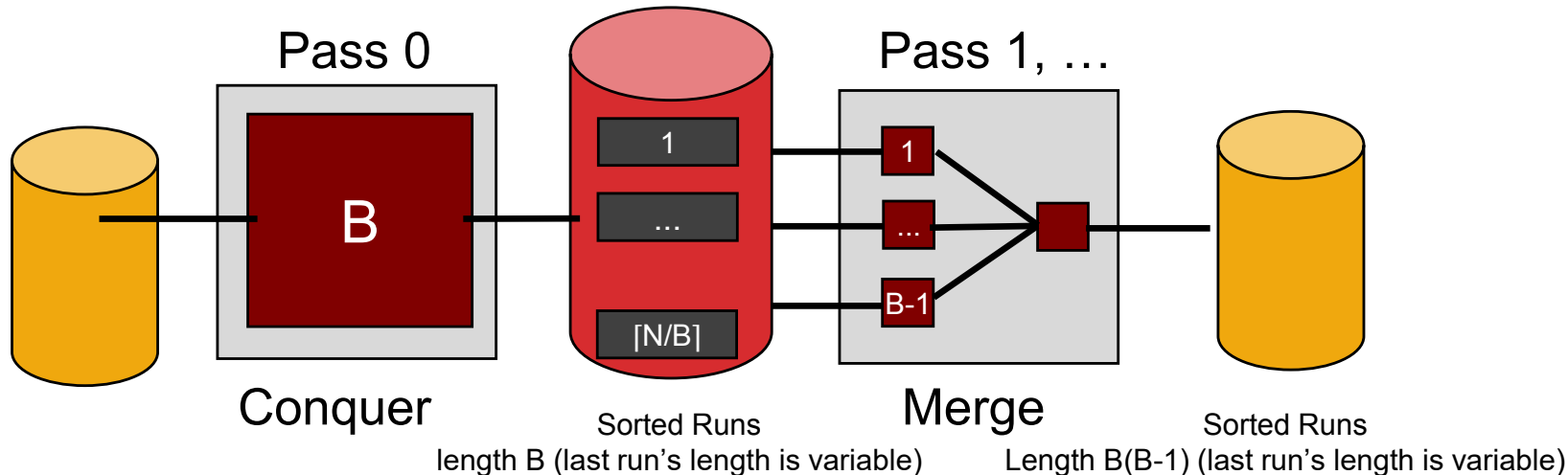
(I/O cost is $2N$ times number of passes)

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

\log fanout \uparrow
pass \downarrow

Memory Requirement for External Sorting

- How big of a table can we sort in two passes?
 - Each “sorted run” after Phase 0 is of size B
 - Can merge up to $B-1$ sorted runs in Phase 1
- Answer: $B(B-1)$. *max 2pass*
 - Sort N pages of data in about $B = \sqrt{N}$ space



Alternative: Hashing

- Idea:
 - Many times we don't require order
 - E.g.: removing duplicates
 - E.g.: forming groups
- Often just need to rendezvous matches
- Hashing does this
 - But how to do it out-of-core??

(but require similar
things to be
put together)

Divide

- Streaming Partition (divide):
Use a hash function h_p to stream records to disk partitions
 - All matches rendezvous in the same partition.
 - Each partition a mix of values
 - Streaming alg to create partitions on disk:
 - "Spill" partitions to disk via output buffers

place for assembling

cause to flow

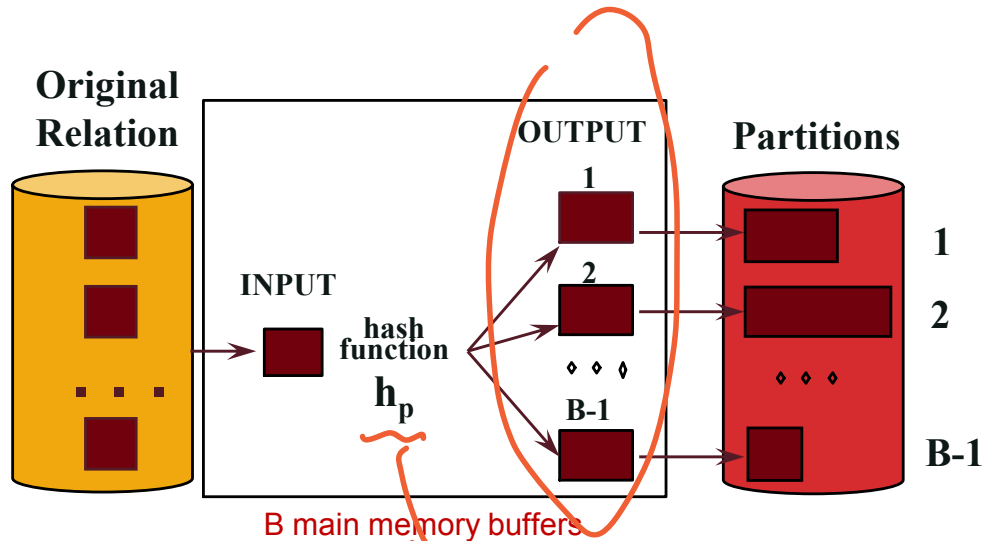
Conquer

- ReHash (conquer):
Read partitions into RAM hash table one at a time, using hash f'n h_r
 - Each bucket contains a small number of distinct values (hash ed)
- Then read out the RAM hash table buckets and write to disk
 - Ensuring that duplicate values are contiguous

Two Phases: Divide

- Partition:
(Divide)

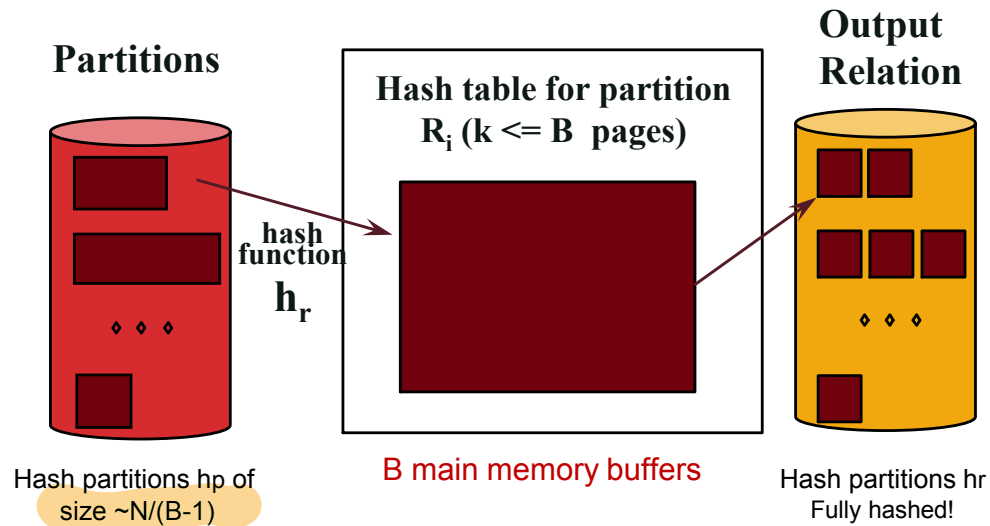
$B-1$ partitions. on scratch disk



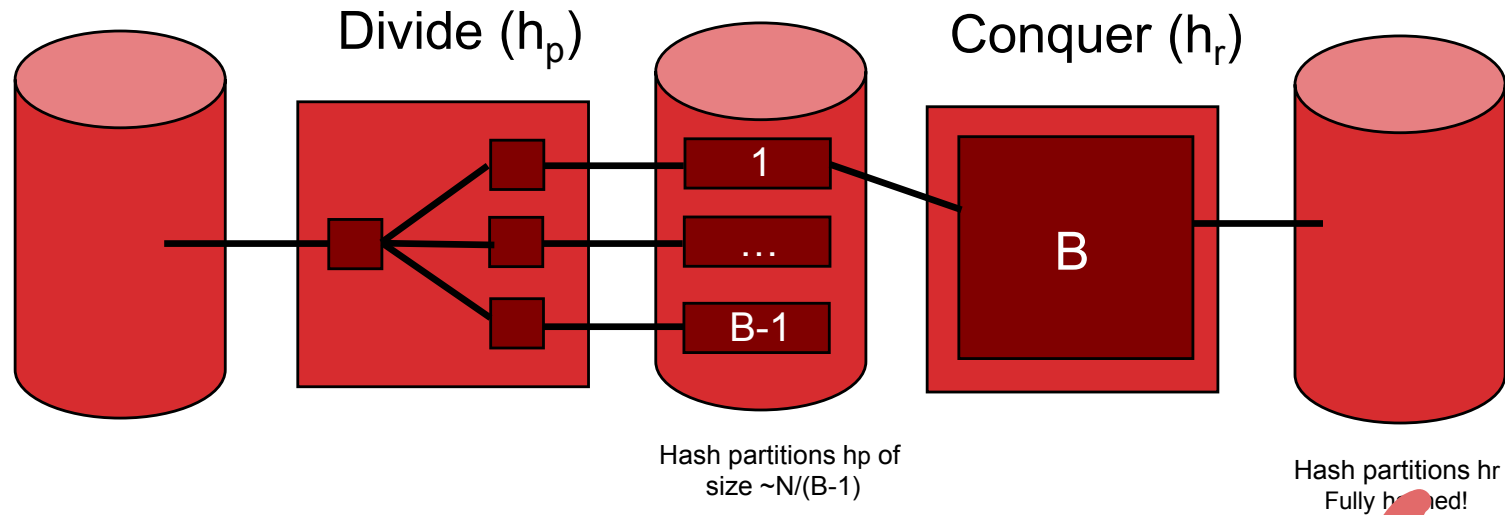
decide which output buffer to go

Two Phases: Conquer

- Rehash:
(Conquer)



Cost of External Hashing



cost = $2 \cdot N \cdot (\text{\#passes}) = 4 \cdot N$ IO's
(includes initial read, final write)



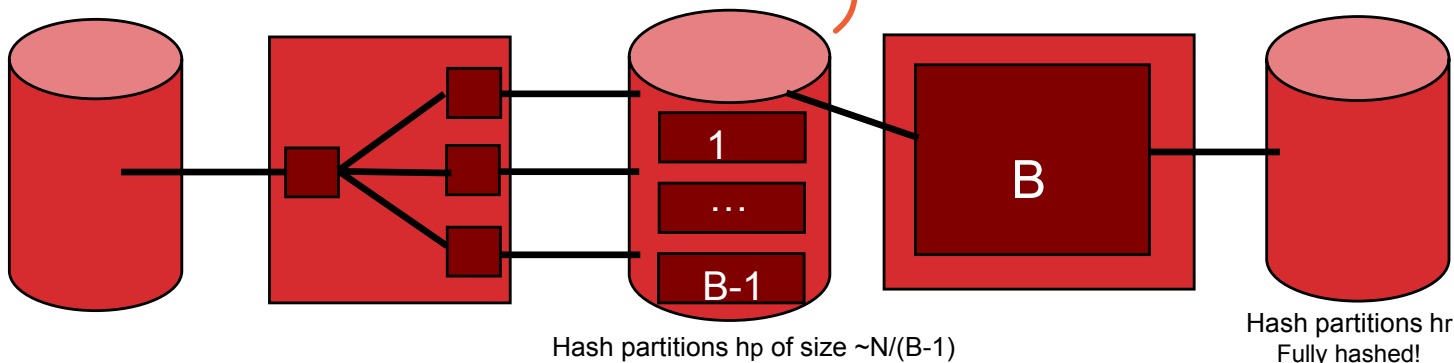
Memory Requirement

- How big of a table can we hash in two passes?

- B-1 “partitions” result from Pass 1
- Each should be no more than B pages in size
- Answer: $B(B-1)$. (block size, often pages)
 - We can hash a table of size N pages in about $B = \sqrt{N}$ space
 - Note: assumes hash function distributes records evenly!

- Have a bigger table? Recursive partitioning!

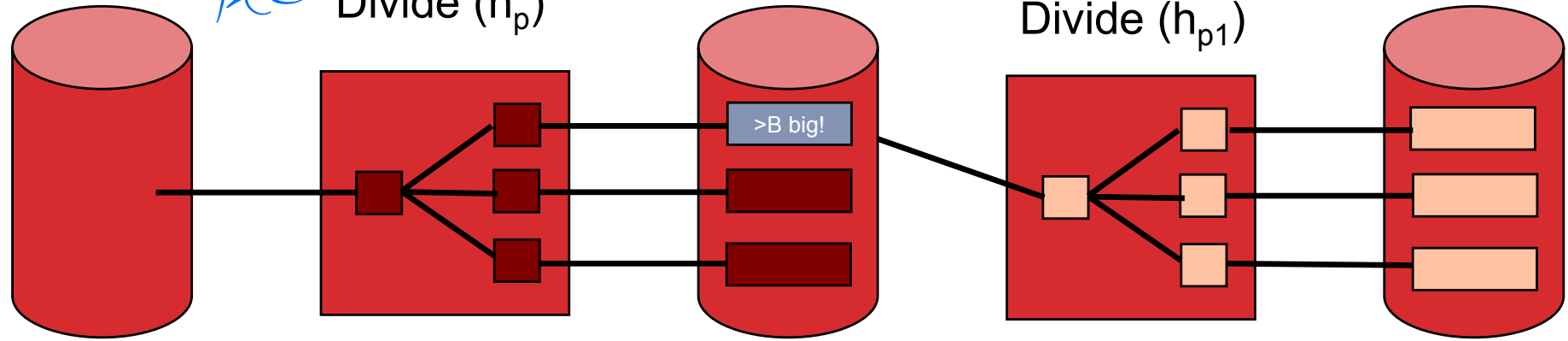
Divide (h_p) $> B(B-1)$ Conquer (h_r)



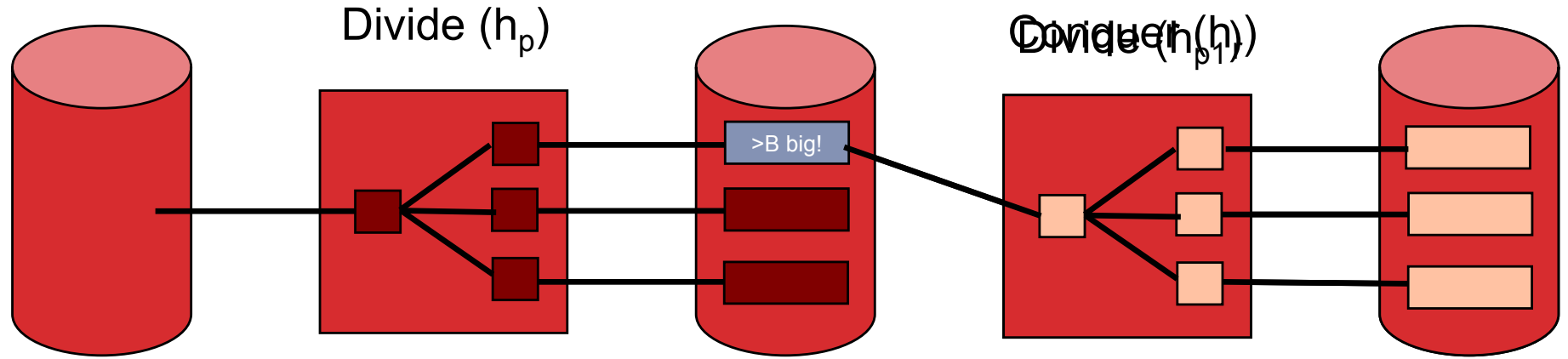
Convert first!

Recursive Partitioning, Pt 1

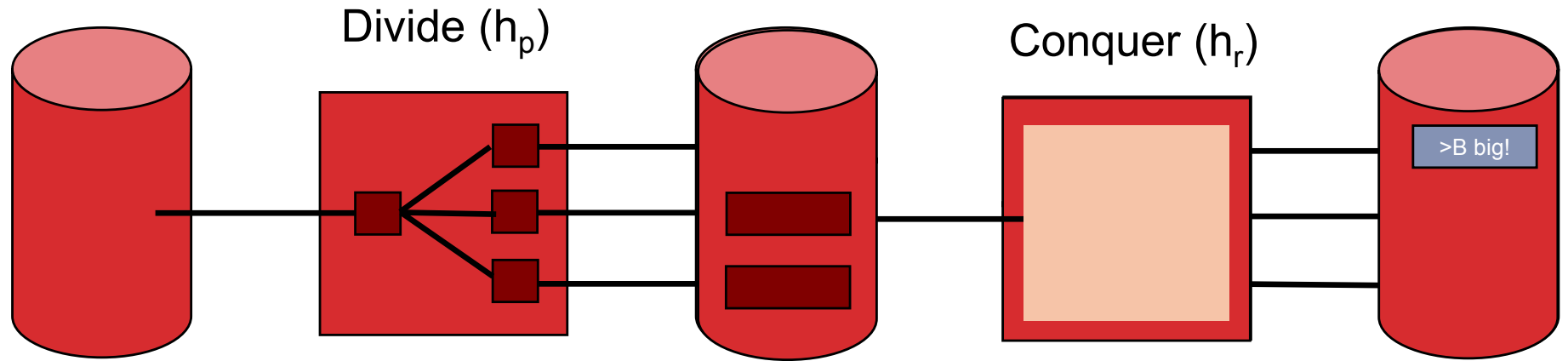
Cost: $2 \times$ block number (# of pages) I/O
additional
Read, write
Divide (h_p)



Recursive Partitioning, Pt 2

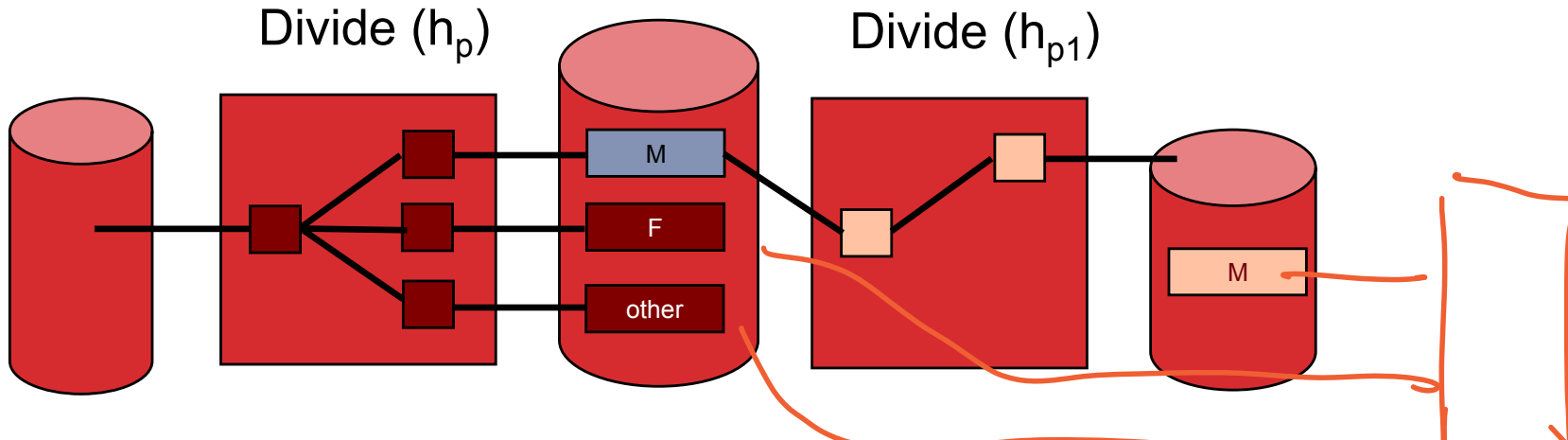


Recursive Partitioning, Pt 3



A Wrinkle: Duplicates

- Consider a dataset with a **very frequent key**
 - E.g. in a big table, consider the *gender* column
- What happens during recursive partitioning?

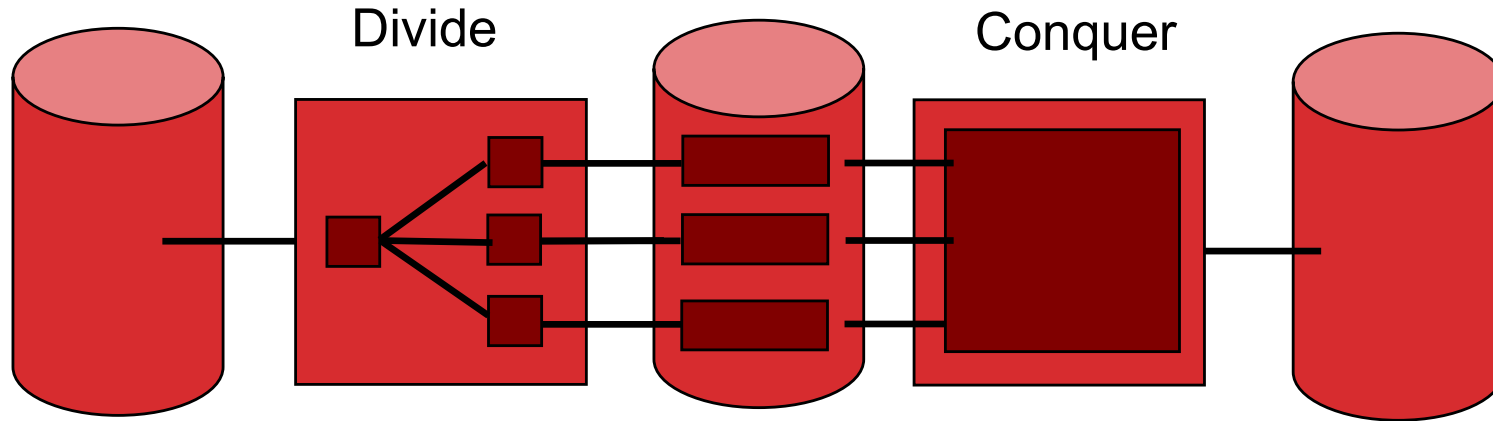




Question...

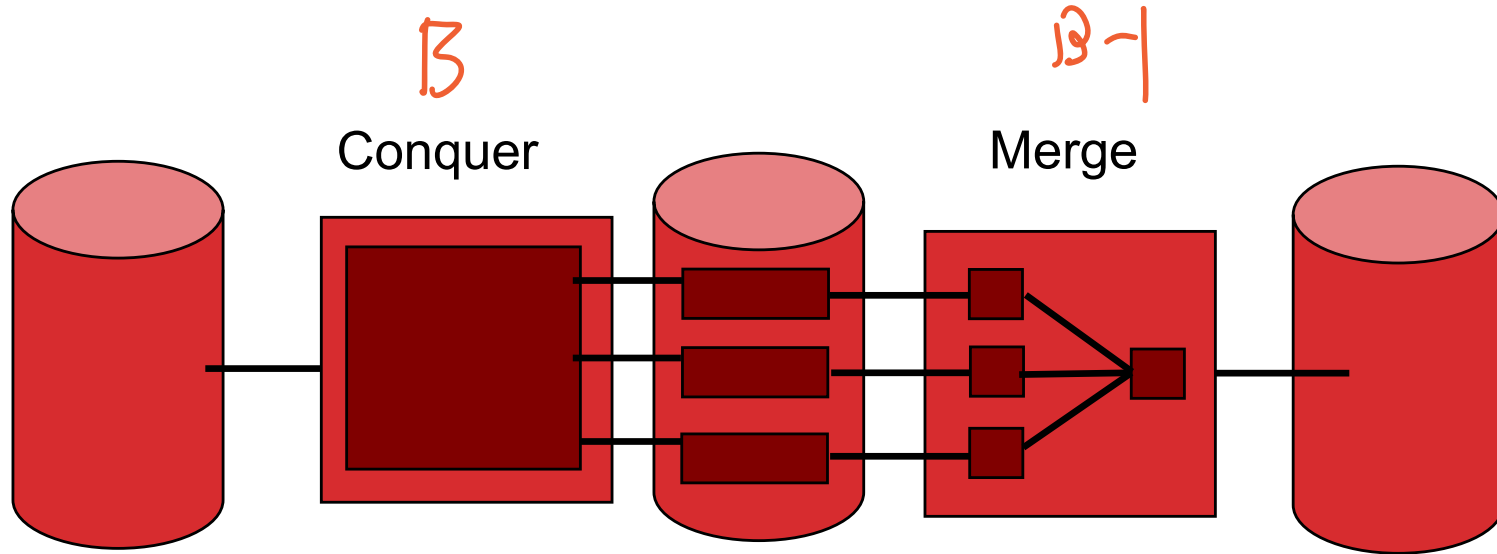
How does external hashing compare
with external sorting?

Cost of External Hashing



cost = $4 \cdot N$ IO's
(including initial read, final write)

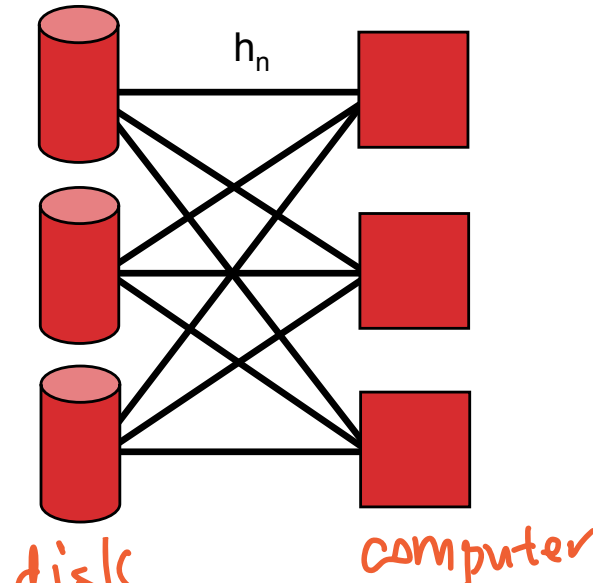
Cost of External Sorting



cost = $4 \cdot N$ IO's
(including initial read, final write)

Parallelize me! Hashing Phase 1

- Phase 1: shuffle data across machines (h_n)
 - streaming out to network as it is scanned
 - which machine for this record?
use (yet another) independent hash function h_n

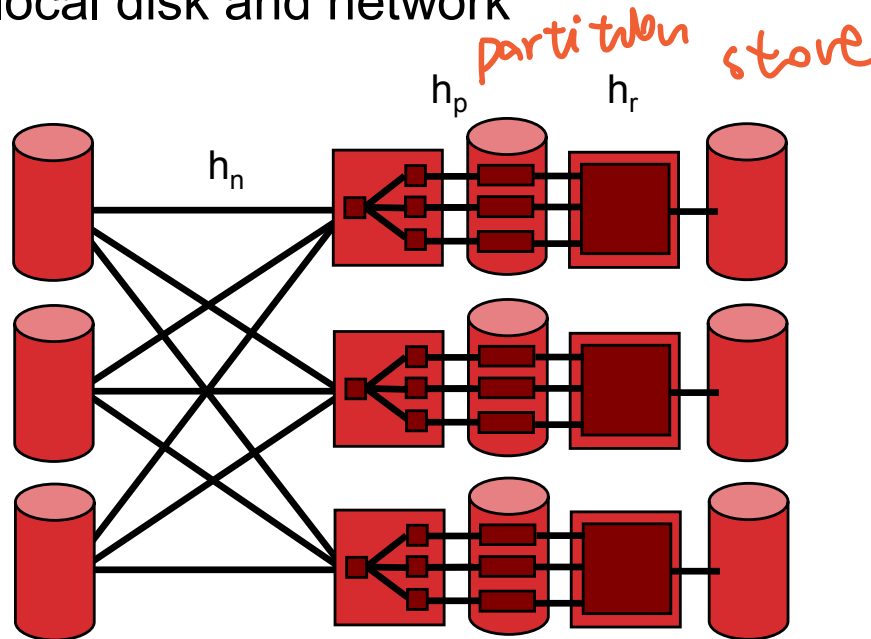


stream out from
disk to PC

$h_{n1} = h_{n2}$
① same hash value
② same machine

Parallelize me! Hashing Phase 2

- Phase 1: shuffle data across machines (h_n)
- Receivers proceed with phase 1 as data streams in
 - from local disk and network



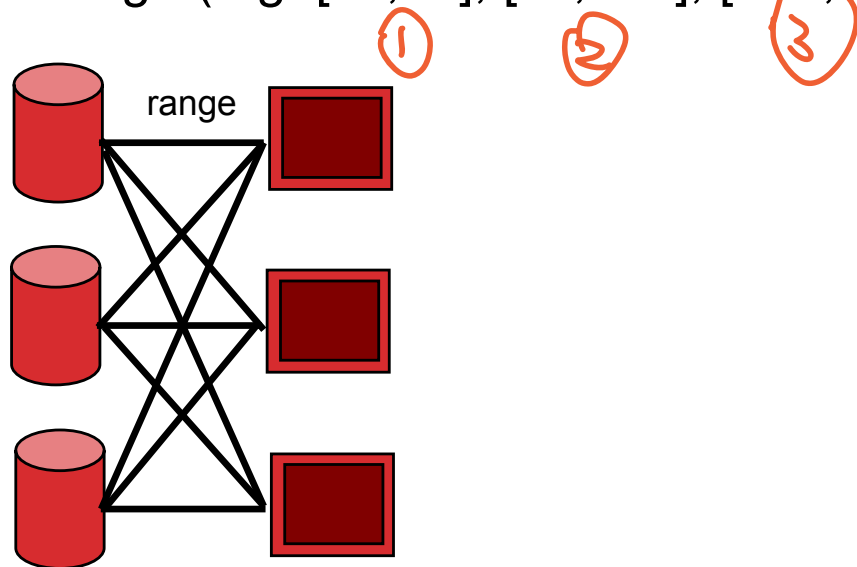
$N = \# \text{ of } PC$
 N times
faster

Parallelize me! Sorting

- Pass 0: shuffle data across machines

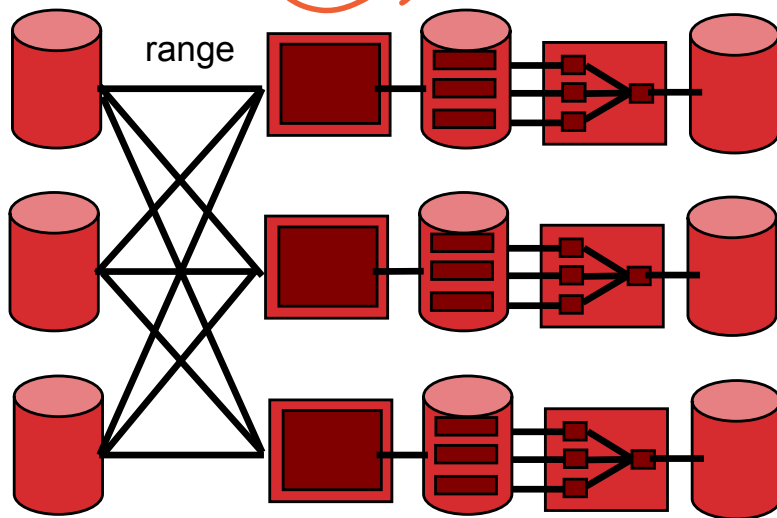
- streaming out to network as it is scanned
- which machine for this record?

Split on **value range** (e.g. $[-\infty, 10]$, $[11, 100]$, $[101, \infty]$).



Parallelize me! Sorting, cont

- Pass 0: shuffle data across machines
- Receivers proceed with pass 0 as the data streams in *size B*
- A Wrinkle: How to ensure ranges are the same #pages?!
- i.e. avoid data skew? *Data > space on PC (B)*



So which is better ??

- Simplest analysis:
 - Same memory requirement for 2 passes
 - Same I/O cost
 - But we can dig a bit deeper...

Goal: estimate distribution of the sorted column

Sorting vs Hashing



- Hashing pros:
 - For duplicate elimination, scales with # of values
 - Delete dups in first pass while partitioning on hp
 - Vs. sort which scales with # of items!
 - Easy to shuffle equally in parallel case
- Sorting pros:
 - Great if we need output to be sorted anyway
 - Not sensitive to duplicates or “bad” hash functions

Summary

- Sort/Hash Duality
 - Hashing is Divide & Conquer
 - Sorting is Conquer & Merge
- Sorting is overkill for rendezvous
 - But sometimes a win anyhow
- Don't forget one pass streaming and double buffering
 - Can “hide” the latency of I/O behind CPU work