CS222/CS122C: Principles of Data Management

UCI, Fall 2019
Notes #11

Join!

Instructor: Chen Li

Schema for Examples

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

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages. (Total cardinality is thus 100,000 reservations)
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages. (Total cardinality is thus 40,000 sailing club members)

Equality Joins With One Join Column

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- In algebra: R $\bowtie$ S. Incredibly common! Must thus be carefully optimized. R $\times$ S is large; so, doing R $\times$ S followed by a selection would be highly inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, **R** is **R**eserves and **S** is **S**ailors.
- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. (We will ignore output costs.)

Simple Nested Loops Join(s)

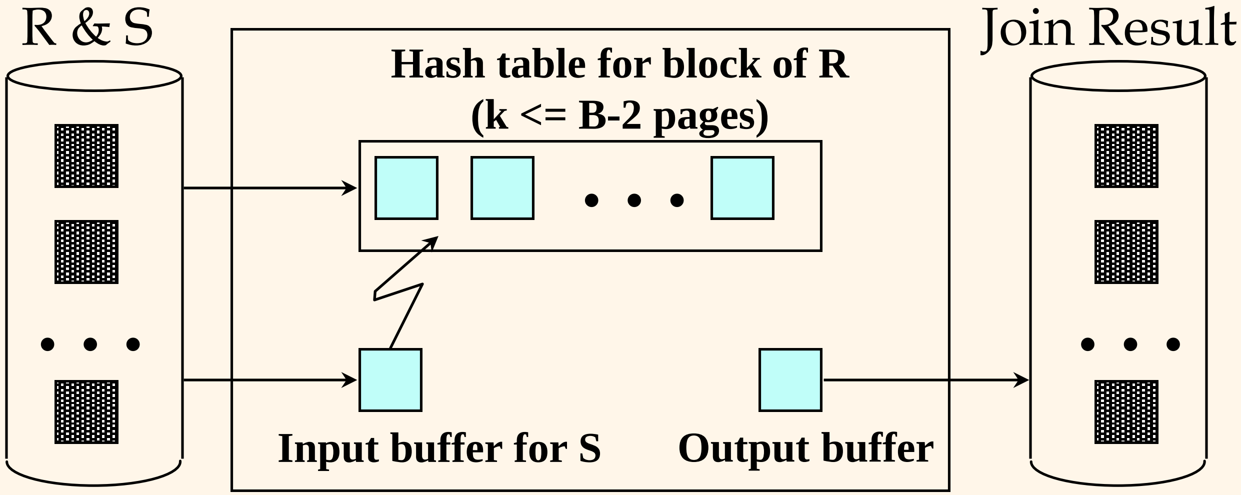
foreach tuple r in R do
    foreach tuple s in S do
        if $r_i == s_j$ then add <r, s> to result

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: $M + (p_R * M) * N = 1000 + (100*1000)*500$ I/Os
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: $M + M*N = 1000 + 1000*500$
  - If smaller relation (S) is outer, cost = 500 + 500*1000

    (Notice that we were essentially wasting an opportunity!)

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use *all* remaining pages to hold a "block" of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks = $\lceil \# \; of \; pages \; of \; outer \; / \; blocksize \rceil$
  - M + M/B * N
- With Reserves (R) as outer, and 100-page blocks of R:
  - Cost of scanning R is 1000 I/Os; there'll be 10 *blocks* total.
  - Per R block, we scan Sailors (S); 10*500 (=5000) I/Os for S.
- With (100-page blocks of) Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os for R.
- (With *sequential reads* considered, analysis changes: may want to divide buffers evenly between R and S.)

Index Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S <u>where</u> r_i == s_j do
        add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((p_R * M) * cost of finding matching S tuples)
- For each R tuple, cost of probing S's index is about 1.2 for hash index, and say 2-4 for B+ tree. Cost of fetching actual S tuples (assuming Alt. (2) or (3) for index data entries) depends on clustering.
  - Clustered (s_j) index: 1 I/O per R tuple (typical); unclustered: 1 I/O per matching S tuple per R tuple.

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 (=100,000) tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os. (Plus 1000 in Reserves "noise".)
- Formula: 1000 + 100 * 1000 * (1.2 + 1)

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 (=40,000) tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on the index's clustering setup.
- Formula: 500 + 500 * 80 * (1.2 + 100,000/40,000)

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

- Sort R and S on the join column, then scan them to do a "merge" (on join column), and output result tuples.
  - Advance scan of R until current R-tuple **>=** current S tuple, then advance scan of S until current S-tuple **>=** current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R scanned once; each S group scanned once per matching R tuple. (Multiple scans of an S group very likely to find all needed pages in buffer pool.)

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: Sort R (roughly M log M) + sort S (roughly N log N) + (M+N)

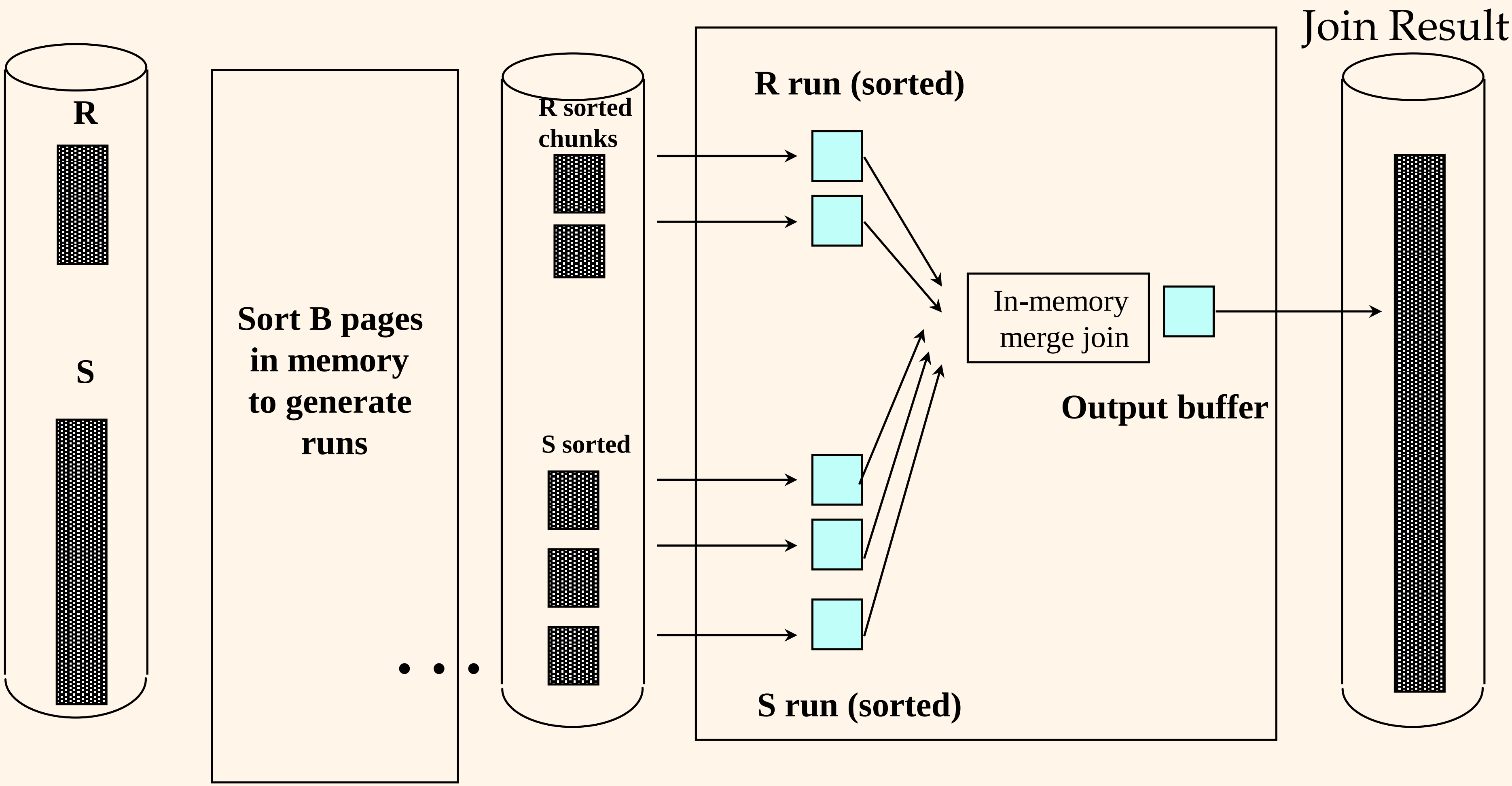
Cost of Sort-Merge Join

- The cost of scanning M+N could be M*N in the worst case
  - When there are many records with the same join value
  - Very unlikely
- With 35 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500 = 5 * (1000 + 500).
- ***Versus*** *BNL cost: 2500 to 15000 I/O range*

Improving Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
- Allocate 1 page per run of <u>each</u> relation, and do a parallel `merge' while checking the join condition. (Else wasteful!)
- Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - Cost: goes down from 7500 to 4500 I/Os (3 * (1000 + 500)).
- With large memory, the cost of sort-merge join, like the cost of external sorting, behaves like it's *linear*.

Improving sort-merge join



Memory requirement: M/B + N/B <= B – 1 So B >= sqrt (M + N)

Improving sort-merge join

First create sorted runs for each table (S & R) .....

sid	sname	rating	age
22	dustin	7	45.0
44	guppy	5	35.0

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
58	rusty	10	35.0

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

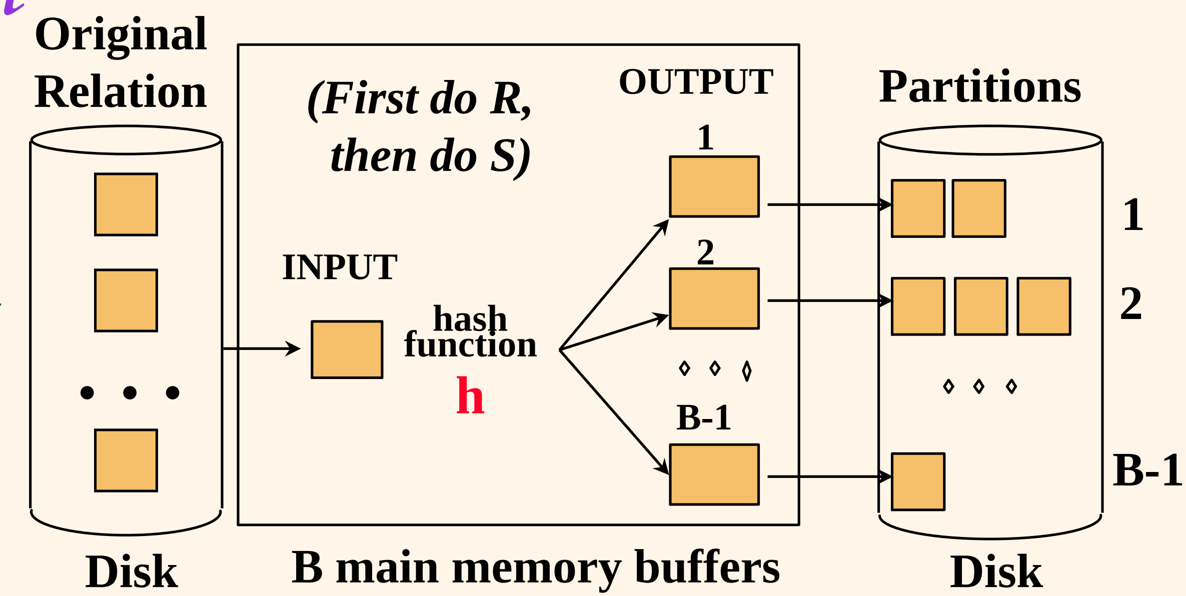
sid	bid	day	rname
28	103	11/3/96	yuppy
31	101	10/10/96	dustin

$\bowtie$

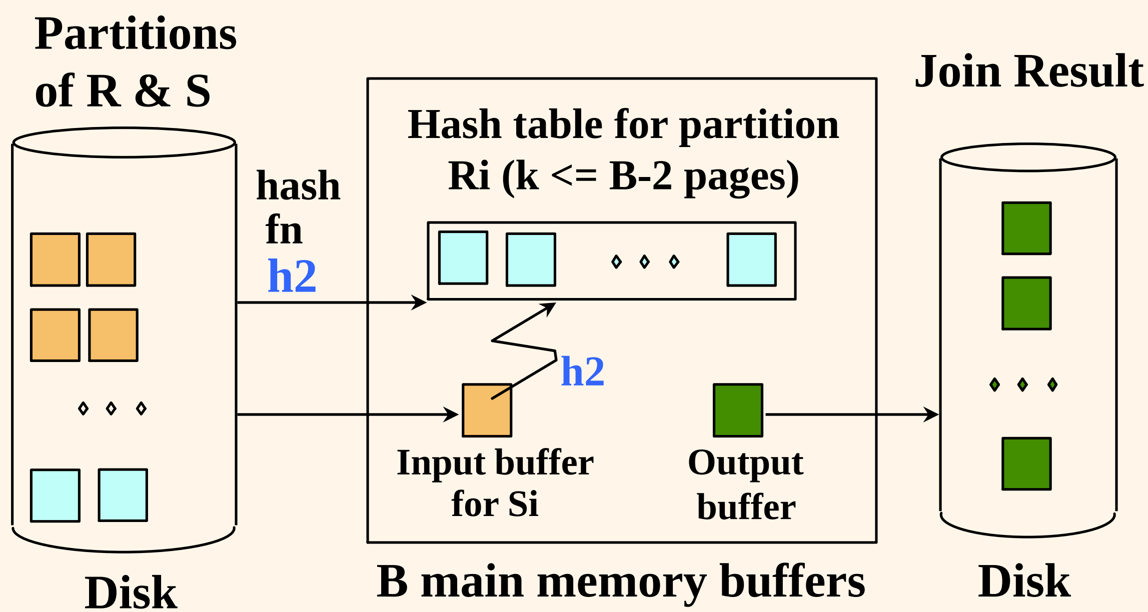
..... then merge S & R runs at the same time as joining them...!

Grace Hash-Join

- Partition both of the relations using a hash function **h**: R tuples in R's partition i will *only* match S tuples in S's partition i.



- Read in one partition of R, hashing it using function **h2** (<> **h**). Scan the matching partition of S, search for its R matches.



Observations on Grace Hash-Join

- #partitions k ≤ B-1 (why?), and B-2 ≥ size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  - k= B-1, and M/(B-1) ≤ B-2, i.e., B must be ≥ sqrt(M)
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can try hash-join technique <u>recursively</u> to do the join of such an R-partition with its corresponding S-partition.

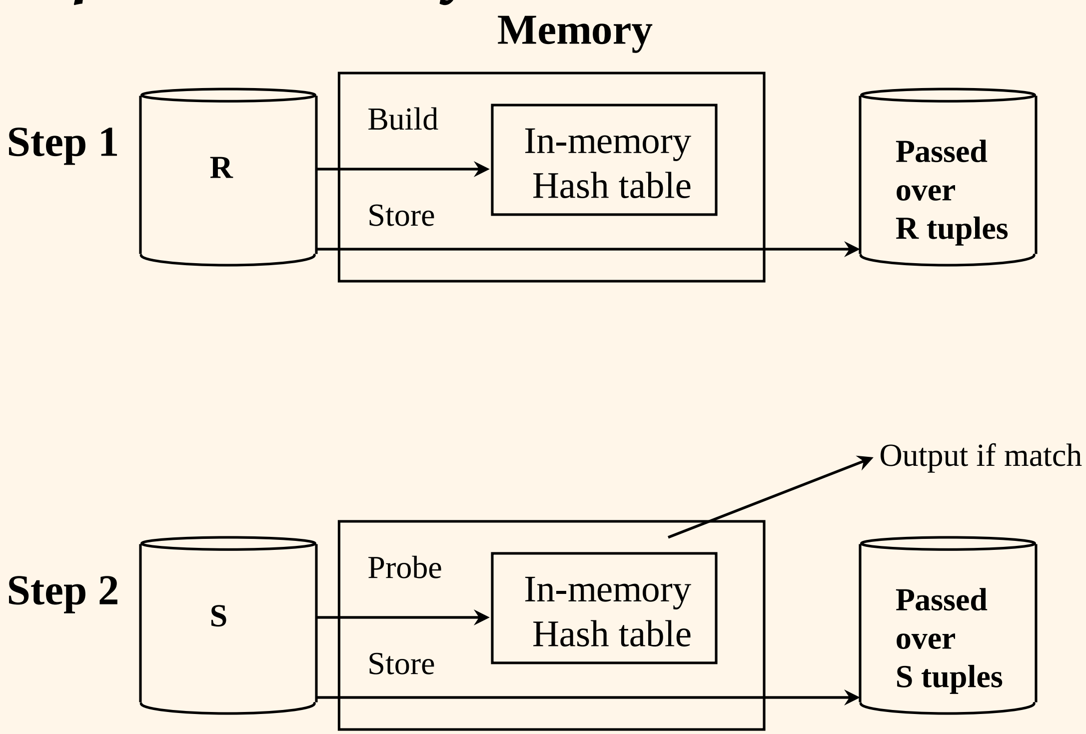
Cost of Grace Hash-Join

- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.
- In our running example, this is a total of 4500 I/Os.
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory, both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly (as it wins by *needing less memory*). Also, Hash Join is neatly parallelizable, as we will see later.
  - Sort-Merge less sensitive to data skew; result is sorted.

Simple Hash-Join (see Shapiro!)

- Grace Hash-Join:
  - Partitioning ("build") does read+write of R and S [2(M+N)]
  - Matching ("probe") does read of R and S; M+N [3(M+N)]
  - *Q:* What if the smaller relation (R) is just *slightly* too big?
- Simple Hash-Join addresses this case:
  - Scan R, using most of memory for a hash table; write overflow part of R to R′(R′s leftovers) on disk
  - Scan S, probe R′s hash table with "most" of S; write overflow part of S to S′ (S′s leftovers) on disk
  - Repeat (recursively!) to join R′ and S′

Simple hash join



Repeat steps 1 and 2 for passed over tuples

"Simple" Hash-Join (see Shapiro!)

first Read

- Cost when |R| ≈ 2B? ➔ half of the pages go to the disk
- Cost: $(M+N) + (.5M+.5N) * 2 = 2(M+N)$

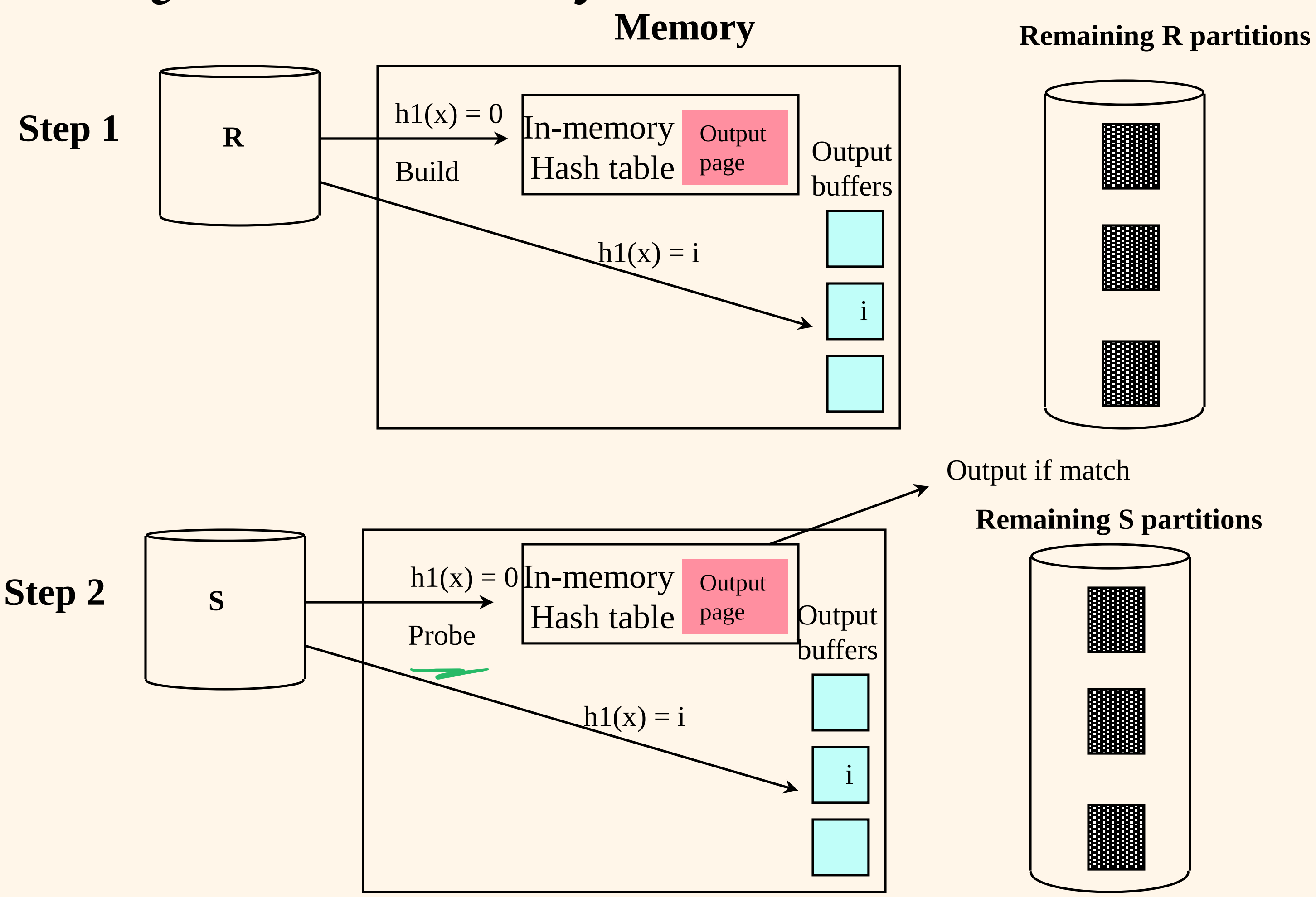
(first write + second read)

- Reason:
  - M + N: each relation needs to be read into memory
  - 0.5M + 0.5N: half of the pages need to be written to disk and read from the disk again (round trip, thus "* 2")

Hybrid Hash-Join (see Shapiro!!)

- Grace Hash-Join wins when both tables are very large compared to our memory allocation B. (Big Data? ☺)
- Simple Hash-Join wins when R almost fits in memory. Q: What do you do when you have two winners, each with a region of superiority? → "Hybrid" Algorithm!
- And thus Hybrid Hash-Join was born…
  - Use a portion of B for an in-memory R hash table
  - Use the rest of B for Grace-style partition buffering
  - Result is that the leftovers are now partitioned!
    - Like Grace HJ when B is small (if no room for a hash table)
    - Like Simple HJ when $B \approx |R| + \varepsilon$ (if only one partition is spilled)

Hybrid hash join



Step 3: process remaining R/S partitions as in the Grace Join

Join with multiple attributes

- Equalities over several attributes (e.g., *R.sid*=*S.sid* AND *R.rname*=*S.sname*):
  - For Index NL, build index on <*sid, sname*> (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort or hash-partition on the combination of the two join columns.

More General (θ) Join Conditions?

- Inequality conditions (e.g., *R.rname* < *S.sname*):
  - For Index NL, need (clustered!) B+ tree index. → scan S for each record of R
    - Range scans on inner; # matches likely much higher vs. equijoins
  - Hash Join, Sort Merge Join simply not applicable.
  - Block NL quite likely to be the best join method here (ditto for other predicates w/non-hashable functions)

Block NL

scan S for all B-2 pages of R,

So Many Joins, So Little Time…!

- Nested Loops Join:
  - Simple NL-Join
  - ***Index NL-Join***
  - Page NL-Join → ***Block NL-Join*** (also works for θ- joins!)
- Sort-Merge Join
  - ***SM-Join*** can avoid sorting R and/or S if ordered; can read and merge run sets from R and S together during the join
- Hash-Join
  - Grace Hash-Join
  - Simple Hash-Join
  - ***Hybrid Hash-Join***