Join Strategies

Chapter 12 and 14

Operator Evaluation

- How to implement common operators?
  - Selection
  - Join
    - Basic strategies
    - Advanced strategies
  - Projection (optional DISTINCT)
  - Set Difference
  - Union
  - Aggregate operators (SUM, MIN, MAX, AVG)
  - GROUP BY

Join Operator

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

- Commercial systems spend a lot of effort optimizing equality joins
  - Why?
  - What is the major source of performance cost when joining two (large) relations?
- Cost Metric: # of I/Os
  - We ignore final output cost since it's the same no matter which algorithm we use

A SQL query walks up to two tables in a restaurant and asks: "Mind if I join you?"

Join Operator

Many different ways of evaluating joins

Sailors

sid	name
1	Lucky
2	Rusty
3	Bob
4	Fred

$\bowtie_{sid = sid}$

Reserves

sid	name
1	100
1	200
3	300
4	200

How would you evaluate this join in memory?

Simple Nested Loops Join

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

- Notation
  - ||R|| = # tuples in R
  - |R| = # pages in R
- How many I/Os ?
  - |R| + ||R|| * |S|

Slightly different notation from textbook!

Page-Oriented Nested Loops

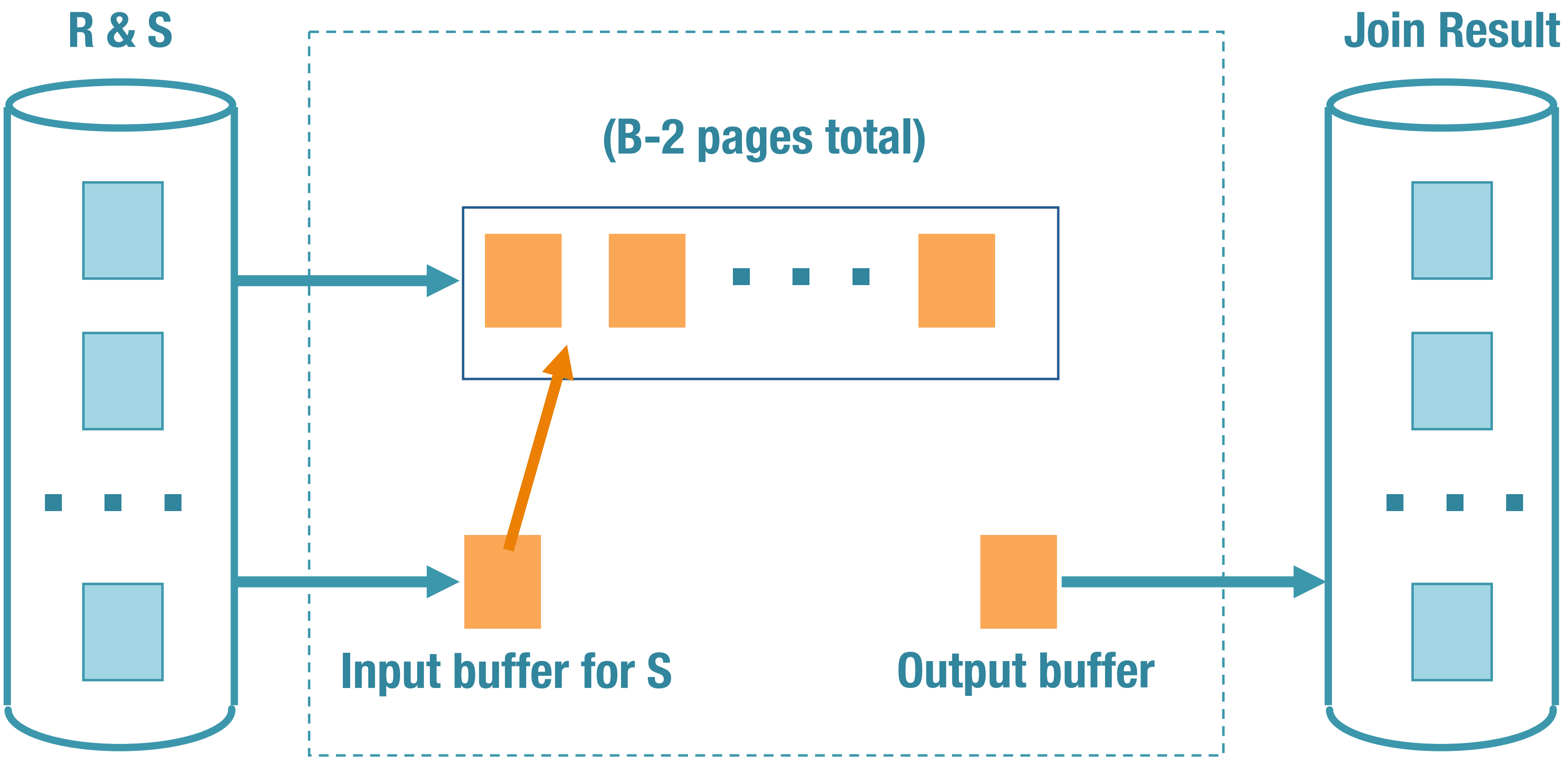
- Page-oriented Nested Loops join:
  - For each <u>page</u> of R, get each <u>page</u> of S, and join
  - How many I/Os?

    |R| + |R| * |S|
  - If S is the outer, then |S| + |R| * |S|

How many buffer pages does this use?

Block Nested Loops

Can we exploit available memory?
Suppose we have B buffer pages available



BNL's Cost Analysis

- Suppose we have B buffer pages available
- Use B-2 pages to hold a block of outer R

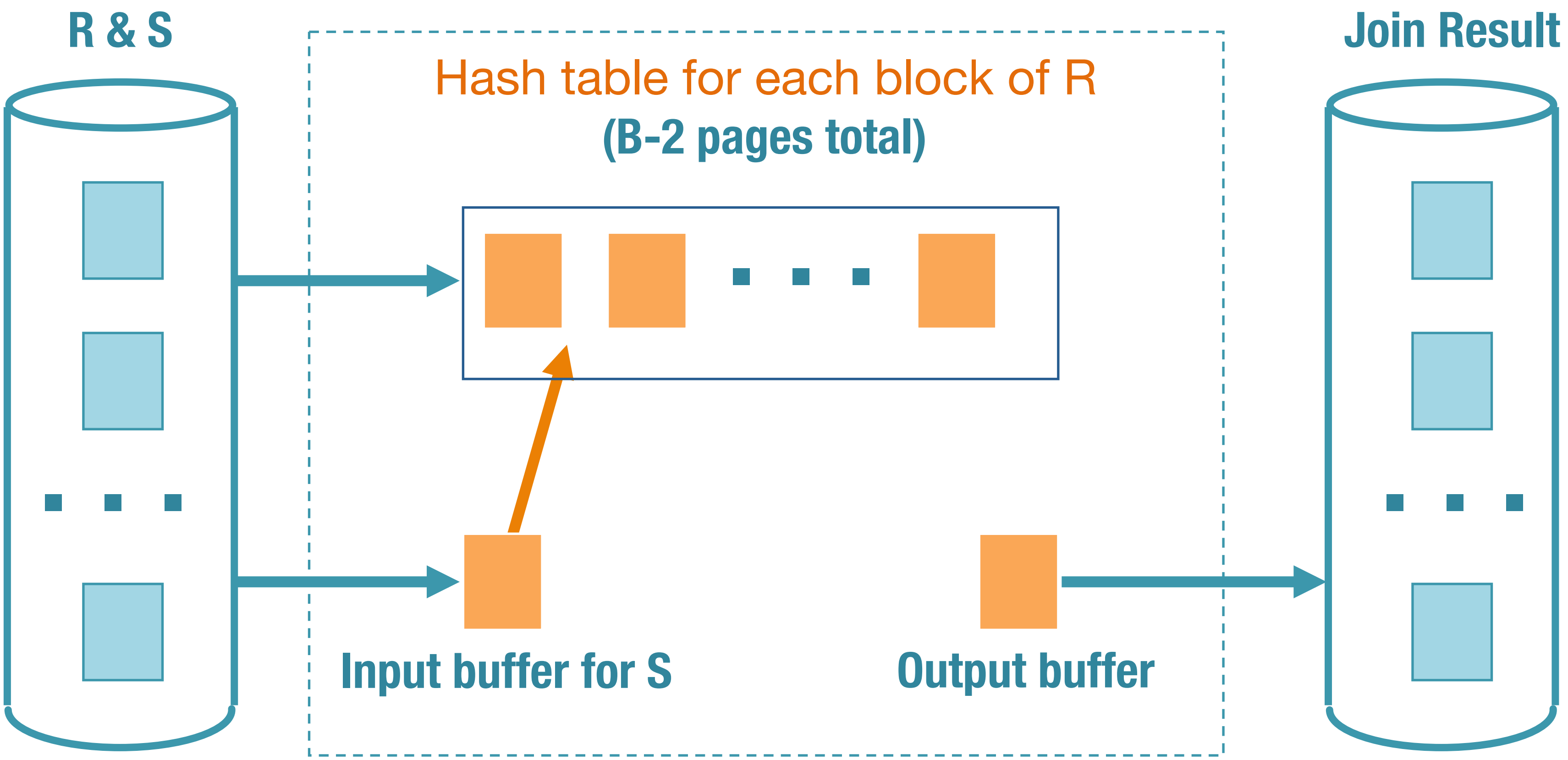
```
For each block of B-2 pages of R do
    for each page of S do{
        for each r in the B-2 R pages do          in memory
            for each tuple s in the S page do
                if r.sid == s.sid  then add <r, s> to result
    }
```

Cost: $$|R| + |S| * \left\lceil \frac{|R|}{B-2} \right\rceil$$

What is the cost if the smaller relation fits entirely in memory?

Block Nested Loops

In-memory hash table: Small overhead for the hash table, but big savings in CPU costs (equality search)



Quiz: PNL vs BNL

|R| = 128 |S| = 64 B = 8
tuples/page for both S and R = 10

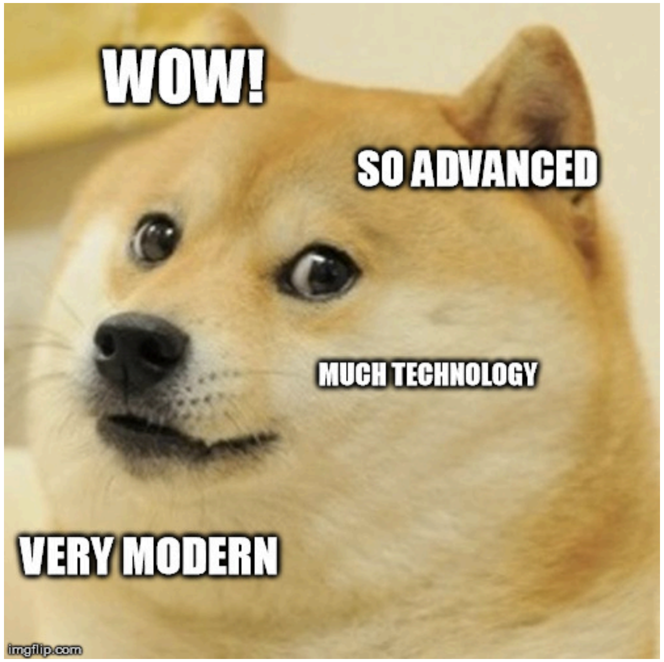
- Page NL
  - Scan outer: 64
  - Join: 64 * 128 = 8192
  - TOTAL: 8256
- Block NL
  - Scan outer: 64
  - Join: $\lceil 64/6 \rceil$ * 128 = 1408
  - TOTAL: 1472

In PNL, which rel. should be the outer?

What about simple nested loops?

Operator Evaluation

- How to implement common operators?
  - ✓ Selection
  - Join
    - ✓ Basic strategies
    - Advanced strategies
  - Projection (optional DISTINCT)
  - Set Difference
  - Union
  - Aggregate operators (SUM, MIN, MAX, AVG)
  - GROUP BY



AdvJ 1: Index Nested Loops

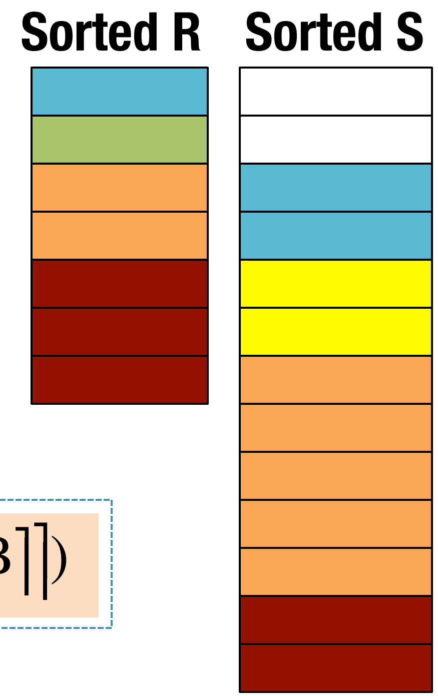
```
foreach tuple r in R do
   Probe Index on S.sid
       foreach matching tuple s do add <r, s> to result
```

- Let's say there is an index available on S
- Use index on join attribute of the inner relation
  - Cost: |R| + (||R|| * cost of finding matching S tuples)
- Cost of finding matching S tuples per R tuple
  - Index cost: 1-2 for hash index
  - Index cost: 2-4 for B+ tree.
  - Record retrieval cost
    - Clustered index: one I/O (typical) for all S tuples per R tuple
    - Unclustered Index: Up to one I/O per matching S tuple

AdvJ 2: Sort-Merge

- Sort R on the join attribute (if necessary)
- Sort S on the join attribute (if necessary)
- Merge

How many buffer pages are used in the merge?



- Sort:

$$2.|R|.(1+\lceil \log_{B-1}\lceil |R|/B\rceil\rceil)+2.|S|.(1+\lceil \log_{B-1}\lceil |S|/B\rceil\rceil)$$

- Merge Cost: (|R|+|S|)
- Merge Worst Case: (|R|*|S|)
  - When?
  - Backups (to disk) needed whenever # tuples in one of S's partitions exceeds buffer size

Quiz: SM, Page NL, Block NL

- $|R| = 128$ $|S| = 64$ $B = 8$
- Find the cost of $R \bowtie S$
- Sort-Merge (use two-way merge sort)
  - Sort R: $2 * 128 (\log_2 128 + 1) = 2048$
  - Sort S: $2 * 64 (\log_2 64 + 1) = 896$
  - Merge: $128 + 64 = 192$
  - TOTAL: 3162

In NL, which rel. is the outer?

- Page NL
  - Scan outer: 64
  - Join: $64 * 128 = 8192$
  - TOTAL: 8256
- Block NL
  - Scan outer: 64
  - Join: $\lceil 64/6 \rceil * 128 = 1408$
  - TOTAL: 1472

Quiz: SM, Page NL, Block NL

- $|R| = 128$ $|S| = 64$ $B = 8$
- Find the cost of $R \bowtie S$
- Sort-Merge (use all the buffers for sort)
  - Sort S: $2*64*(1+\lceil \log_7 \lceil 64/8 \rceil \rceil) = 384$
  - Sort R: $2*128*(1+\lceil \log_7 \lceil 128/8 \rceil \rceil) = 768$
  - Merge (input): 64+128 = 192
  - TOTAL: 1344

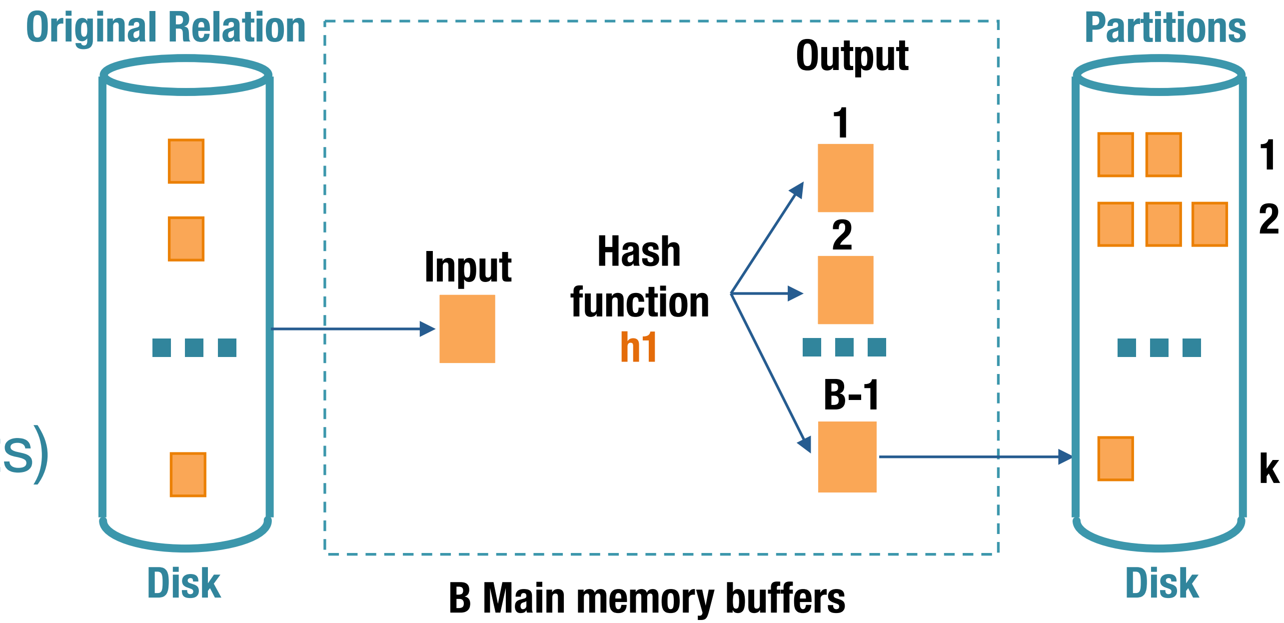
$\log_7 8 = 1.07$
$\log_7 16 = 1.42$

- Page NL
  - Scan outer: 64
  - Join: 64 * 128 = 8192
  - TOTAL: 8256
- Block NL
  - Scan outer: 64
  - Join: $\lceil 64/6 \rceil * 128 = 1408$
  - TOTAL: 1472

AdvJ 3: Grace Hash Join

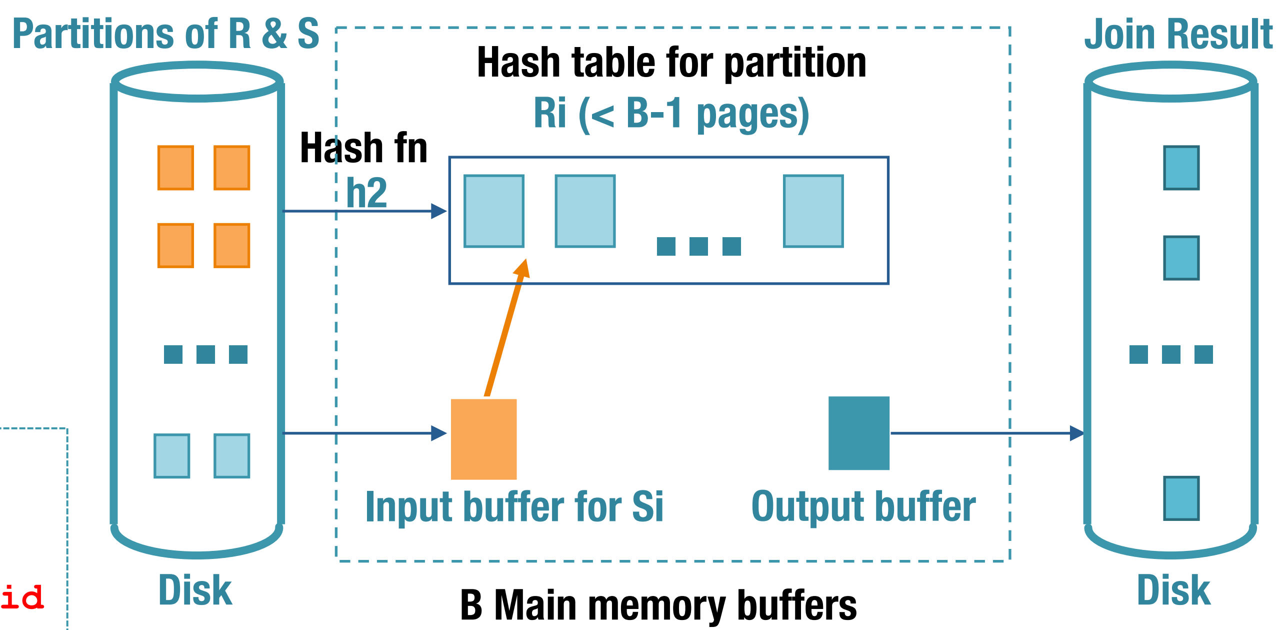
- Step 1: (Partition) Hash both relations, R and S, on the join attribute, producing k disk-based partitions
  - Guarantees that R tuples can only join with S tuples in the same partition (e.g. R1-S1, R2-S2…)
- Step 2: (Probe) Read in complete partition of (smaller relation) R, and scan S partition for matches.
  - Use a different hash function h2 to reduce CPU cost of matching

Step 1: Partition



- k = B-1 (# of buckets)
- h1 ≠ h2

Step 2: Probe

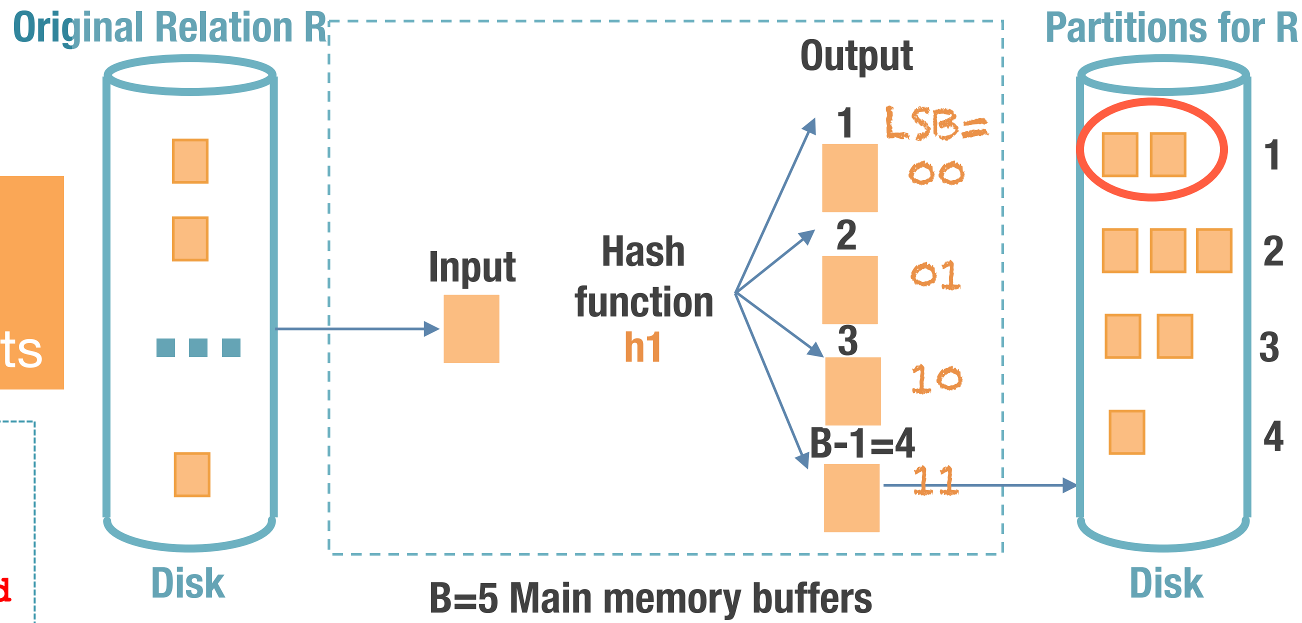



```
SELECT  *
FROM    Reserves R,
        Sailors S
WHERE   R.sid = S.sid
```

Step 1: Partition

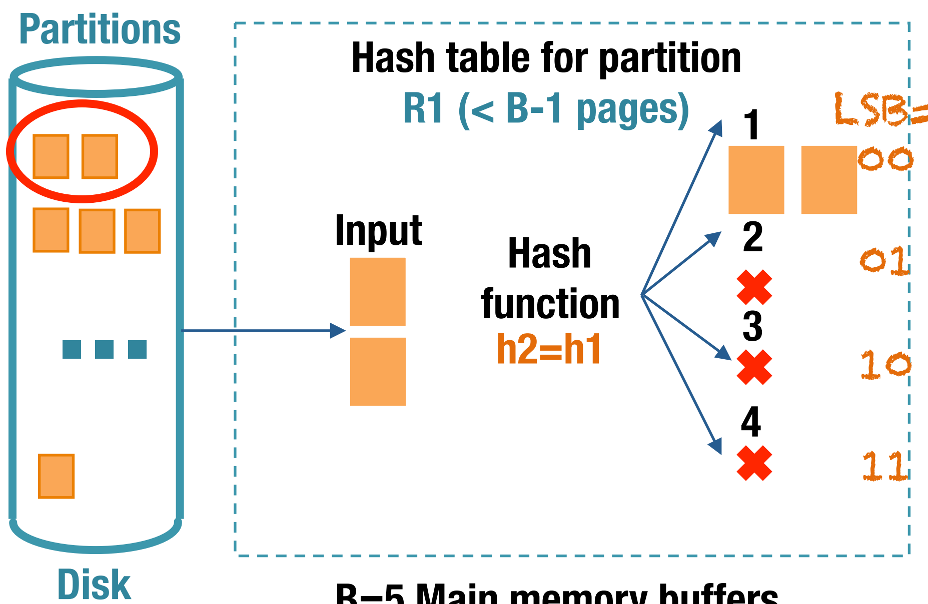
Can h1 = h2?
e.g. h1(sid) = m
Least Significant Bits

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



Part of Step 2: h2 in-memory

Possible sids in partition 00:
$4=(100)_2$
$12=(1100)_2$
$20=(10100)_2$
$32=(100000)_2$ ...

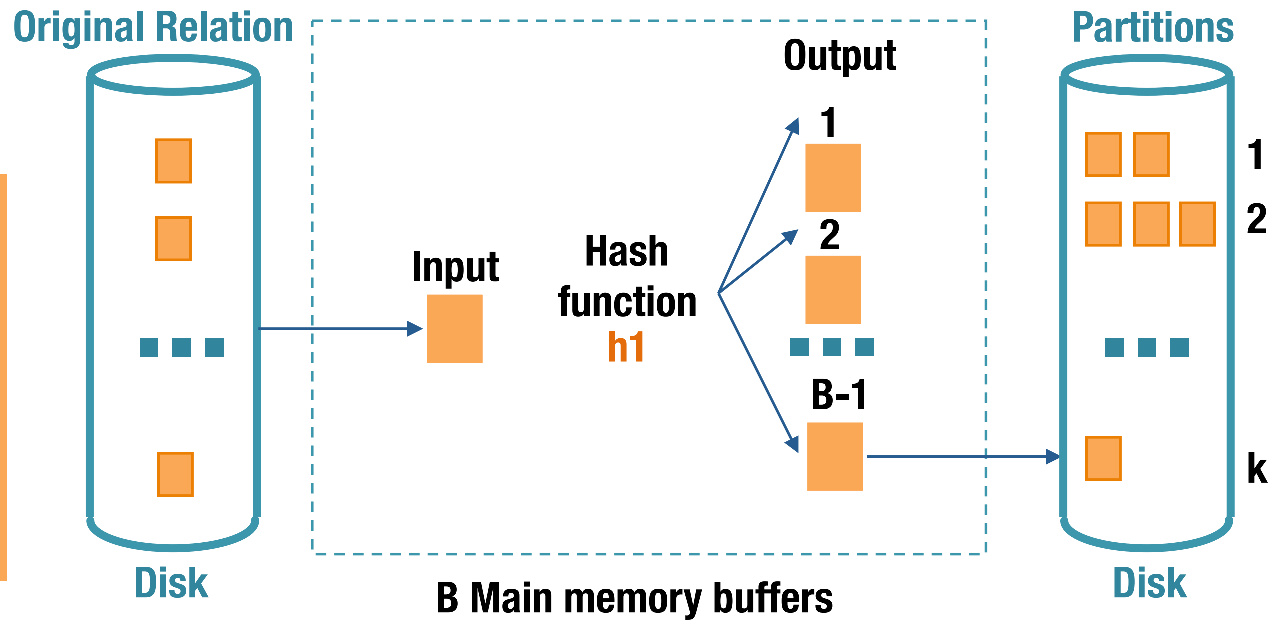


No! The records will be assigned to one bucket (same as before).

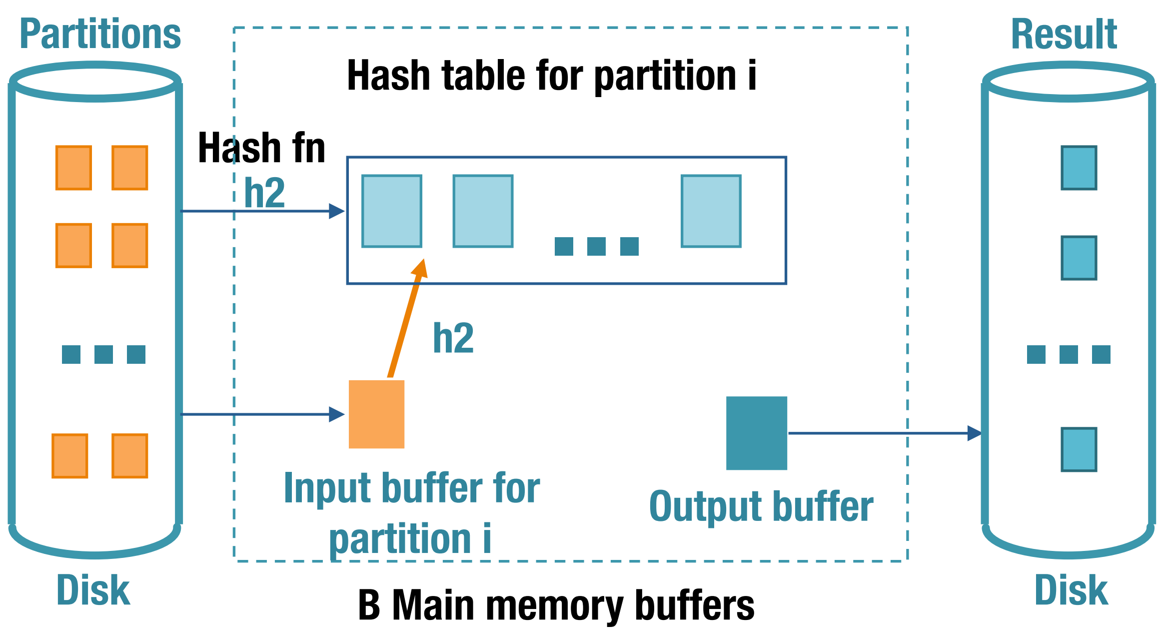
example

Step 1: Partition

What if the hash table for a partition overflows, i.e. can't fit in memory?



Step 2: Probe



Recursively apply hash-based projection technique to handle partition overflow problem

Cost Analysis: Grace Hash Join

- Partition phase, read+write both relations; 2(|R|+|S|)
- In probe phase,
  - Read each partition of relation R once:
    - For that read the corresponding partition of S once
  - Total cost for all partitions: |R|+|S| I/Os.
  - Best-case scenario
  - Assumes each R partition fits in memory in probe phase

Grace hash join: 3(|R| + |S|) I/Os

- The purpose of h2: reducing CPU costs
- Other variants of hash join exist

How to Sort-Merge in 3(|R|+|S|) I/Os

- Before:
  - External sorting: $2.|R|.(1+\lceil \log_{B-1}\lceil |R|/B \rceil \rceil) + 2.|S|.(1+\lceil \log_{B-1}\lceil |S|/B \rceil \rceil)$
  - Merge cost (no backups) = |R|+|S|
  - Merge cost (backups) = |R| * |S|
- **Can be done in 3(|R|+|S|):**
  - When larger relation |S| ≤ B, i.e. each relation fits in memory
- **Rationale**:
  - R fits in memory, sort in 2*|R| I/Os
  - S fits in memory, sort in 2*|S| I/Os
  - Assuming no backups are required, merge would take only |R|+|S|

How to Sort-Merge in 3(|R|+|S|) I/Os

- Before:
  - External sorting: $2.|R|.(1+\lceil \log_{B-1}\lceil |R|/B \rceil \rceil) + 2.|S|.(1+\lceil \log_{B-1}\lceil |S|/B \rceil \rceil)$
  - Merge cost (no backups) = |R|+|S|
  - Merge cost (backups) = |R| * |S|
- **Can be done in 3(|R|+|S|):** square root of |S| fits in memory
  - When larger relation $|S| \leq B^2$, i.e. ~~each relation fits in memory~~
- **Example:** Assume |R| = 400, |S|=10,000 and B = 100
  - Use replacement sort to produce runs of 2*M=2*(B-2)=196 pages long ≈200=2*B for each relation.
  - Then we have at most |R|/(2B)≈2 runs of R and |R|/(2B)≈50 runs of S.
  - Note that since $|R| \leq |S| \leq B^2$ we have $|R|/(2B) \leq B/2$ and $|R|/(2B) \leq B/2$
  - Since total # of runs ≤B, we can read in memory one page per run (here 52 pages), merge, and apply join condition on-the-fly + one output page

Explanation: Why $|S| \leq B^2$ for SM join?

- **Assumption**: S is the largest relation
- The SM join algorithm that gives us the 3(|R|+|S|) IO cost does the following:
  - Replacement sort for R to produce runs of ~2M = 2(B-2) ≈ 2.B pages
  - Replacement sort of S to produce runs of ~2M = 2(B-2) ≈ 2.B pages
  - Now each run is sorted, but each relation is NOT sorted globally. For example 2 of the runs of relation R might look like:

Run 1: [5, 10] [30, 42]

Run 2: [1, 4] [25, 56] [98]

Explanation: Why $|S| \leq B^2$ for SM join?

- **Assumption**: S is the largest relation
- The SM join algorithm that gives us the $3(|R|+|S|)$ IO cost does the following:
  - Replacement sort for R to produce runs of ~2.M = 2.(B-2) ≈ 2.B pages
  - Replacement sort of S to produce runs of ~2.M = 2.(B-2) ≈ 2.B pages
  - Now each run is sorted, but each relation is NOT sorted globally.
  - To do the global sorting + join, we will need to bring in memory at least the first page of each run of R and S. How many runs does S (the largest relation) have? Let's call the quantity r_S.

$$r_S = \text{(\# of pages of S) / (avg run length)} = |S| / (2.B) \quad (1)$$

  - Relation R is smaller, so it will have fewer runs. As we said, we want the first page of each run to fit in memory. This means that we want:

$$r_R + r_S \leq B.$$

Explanation: Why $|S| \leq B^2$ for SM join?

- As we said, we want the first page of each run to fit in memory. This means that we want:

$$r_R + r_S \leq B \quad (2)$$

- In the worst case R and S are of the same size, so we would have the same number of runs for both relations. In this case, relation (2) becomes:

$$2.r_S \leq B$$

- By substituting Eq. (1), we obtain:

$$2.|S| / (2.B) \leq B \Rightarrow$$

$$|S| \leq B^2$$

So, if $\sqrt{|S|}$ fits in memory, we can run the optimized SM join algorithm. ☐

Reminder: Replacement Sort (for Pass 0)

- Start by reading a page from file into the input buffer
- Copy records from input buffer to current set
- Repeatedly pick smallest value from current set that is greater than largest value in output buffer
  - Write to output buffer (run). If buffer full, output
- Start a new run when no value in current set is larger than all values in output

On average, produces runs of size 2M, i.e. 2*(B-2) pages.

Join Algorithms: A Comparison

- Hash-Join vs. Sort-Merge Cost:
  - Sort-merge can be optimized to $3(|R|+|S|)$ I/Os
    - When larger relation $|S| \leq B^2$
    - **Memory requirements:** larger relation $|S| \leq B^2$
  - Hash join costs $3(|R|+|S|)$ I/Os
    - If each partition of the smaller relation fits in memory
    - **Memory requirements:** Smaller relation $|R| \leq B^2$

- Hash Join superior if relation sizes differ greatly
- Hash Join is highly parallelizable
- Hash join worse if partitioning is skewed
- Sort-Merge results already sorted (if matters)

General Join Conditions

- Equalities over several attributes is doable with all Joins!
  e.g. `R.sid=S.sid AND R.rname=S.sname`
  - Block Nested Loop
    - always works
  - Index Nested Loop:
    - index on *<sid, sname>* or sid or sname.
    - Usually more I/Os than BNL
  - Sort-merge join:
    - sort Reserves on <sid, rname> and Sailors on <sid, sname>
  - Hash join:
    - hash Reserves on <sid, rname> and Sailors on <sid, sname>

General Join Conditions

- Inequality conditions (e.g. `R.rname < S.sname`):
  - Block Nested Loop: still works
  - For Index Nested Loop, needs B+ tree index
    - Usually worse than Block Nested Loops
  - Sort-Merge and Hash-Join are not applicable

Summary: Join Strategies

- Basic Join Strategies:
  - Simple nested loops
  - Page nested loops
  - Block nested loops
- Advanced Join Strategies:
  - Index Nested Loops
  - Sort-Merge
  - Hash-Join

Optional Exercises

- 12.1 (1-4), 12.3, 12.5
- 13.1, 13.3
- 14.1 (2, 3, 4, 6, 7, 8, 9, 10), 14