CSE 541: Database Systems I

Query Processing – Join Operations

Slides Credit: Tianzheng Wang

Relational Algebra Operators

- Selection (σ)
  - Select a subset of rows from a table (relation)
- Projection (π)
  - Pick a subset of columns
- Set difference
  - S1 – S2: return tuples that appear in S1 but not in S2
- Union
  - S1 U S2: return tuples in S1 and S2
- Aggregation
  - Operations such as SUM, MIN and GROUP BY
- Join
  - Combine two relations based on some conditions

Join Operators

Goal: join two relations R and S based on a predicate

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

Reserves $\bowtie_{\text{Reserves.sid = Sailors.sid}}$ Sailors

Algorithms:

- Nested loops join
  - Page nested loops join
  - Block nested loops join
  - Index nested loops join
- Sort-merge join
- Grace hash join

Nested Loop Join

Basic idea: scan outer relation R, for each tuple in R, scan the entire inner relation S

```
foreach tuple r in R do:
  foreach tuple s in S do:
    if join-condition(ri, sj) then add <r, s> to
result
```

"tuple-at-a-time"

- Desired: low cost (fewer I/O operations)

Cost (number of I/Os): $M + \boxed{pR * M} * N$

- N I/Os to scan S once (S stored in N pages)
- S scanned for pR * M times

M I/Os to scan the entire R (R stored in M pages)

Number of records in R (pR records per page)

Nested Loop Join

Cost (number of I/Os): M + pR * M * N

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

Reserves table:

- Tuple size: 40 bytes
- 100 tuples per page
- 1000 pages in total

Sailors table:

- Tuple size: 50 bytes
- 80 tuples per page
- 500 pages in total

- If R = Reserves, S = Sailors, then M = 1000, pR = 100, N = 500
  - Cost = **1000 + 100 * 1000 * 500** = 1000 + 5 * 10^7 I/Os
  - Assuming 5ms per I/O ➔ about 70 hours to finish!
- What if we change the join order so Sailors becomes the outer table?
  - M = 500, N = 1000, pR = 80, Cost = **500 + 80 * 500 * 1000** I/Os
  - Smaller than using Reserves as outer relation

➔ Should choose the smaller relation as the outer relation

Page Nested Loop Join

Using the smaller table as the outer relation reduces cost, but not significantly

Solution: scan the inner relation for each page (instead of each tuple) of the outer relation ("page-at-a-time")

```
foreach rpage in R do:
  foreach spage in S do:
    foreach tuple r in rpage do:
      foreach tuple s in spage do:
        if join-condition(ri, sj) then add <r, s> to result
```

Cost (number of I/Os): M + M * N

M I/Os to scan the entire R
(R stored in M pages)

Scan S for each page in M

- If R = Sailors, S = Reserves, total cost = 500 + 500 * 1000 I/Os
  - Much lower than tuple-at-a-time

Block Nested Loop Join

"Block" at a time

- A block may include many pages,
- Read in a block of outer relation each time

Suppose there are B pages in the buffer pool:

```
foreach rblock of B-2 pages in R do:
  foreach spage in S do:
    for all matching tuples in spage and rblock do:
      add <r, s> to result
```

Note: block size = **B – 2**

- One remaining page for writing out results
- One remaining page for reading in the inner relation

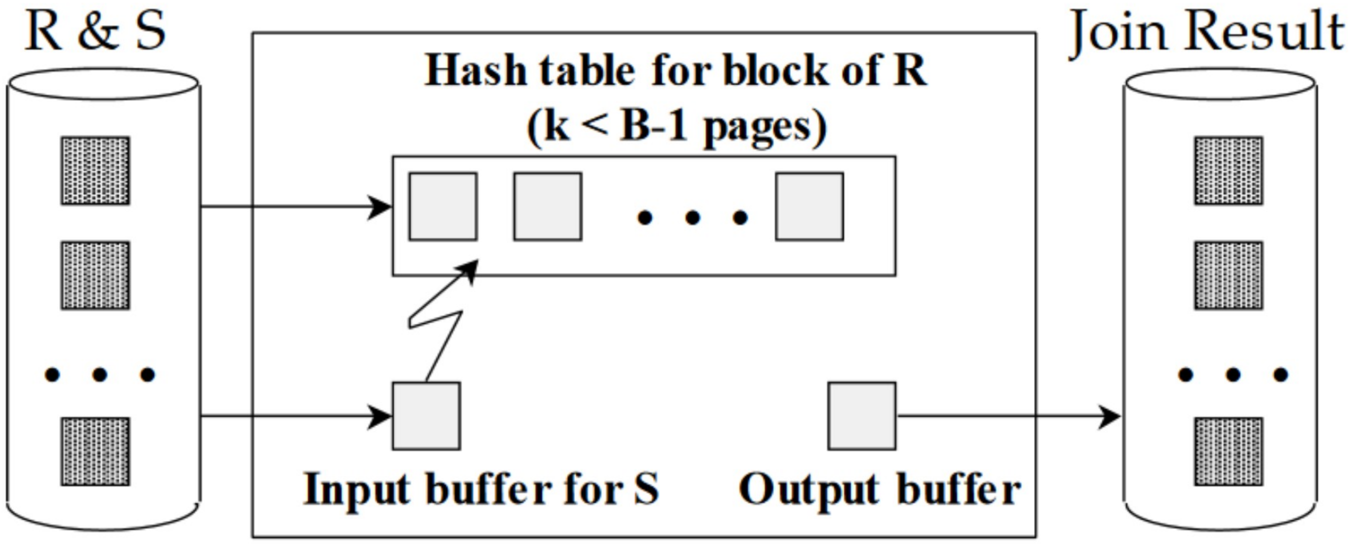
Cost: $M + \lceil M / (B - 2) \rceil * N$

M I/Os to scan R

Number of times needed to scan N

Block Nested Loop Join Refinements

- Build a main-memory hash table for outer relation block
    - Take each S-tuple and query the R-block hash table in memory



    - The hash table occupies slightly larger space then the block itself, but reduces complexity in finding matching pairs
- If there is enough memory: build a hash table for the entire inner relation ➔ Cost = M + N

Index Nested Loop Join

If there is an index on the join column of one relation, make it the inner and exploit the index

- E.g., join R and S where R's column i == S's column j
  - There is an index on S's join column S_j
  - Use R's join column as the search key to look for matching tuples in S

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

Look up r_i in the the index on S

Cost: M + pR * M * cost to find matching tuples in S (using index)

M I/Os to scan R (outer relation)

Number of records in R (i.e., number of lookups in S's index)

- Cost of accessing index may vary
  - What's stored in data entries
  - Clustered vs. unclustered

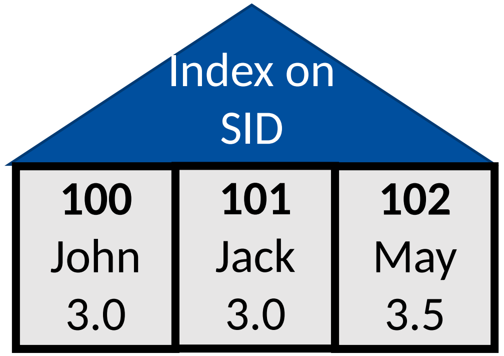
Recap: Index Data Entries

Alternative 1: Actual record

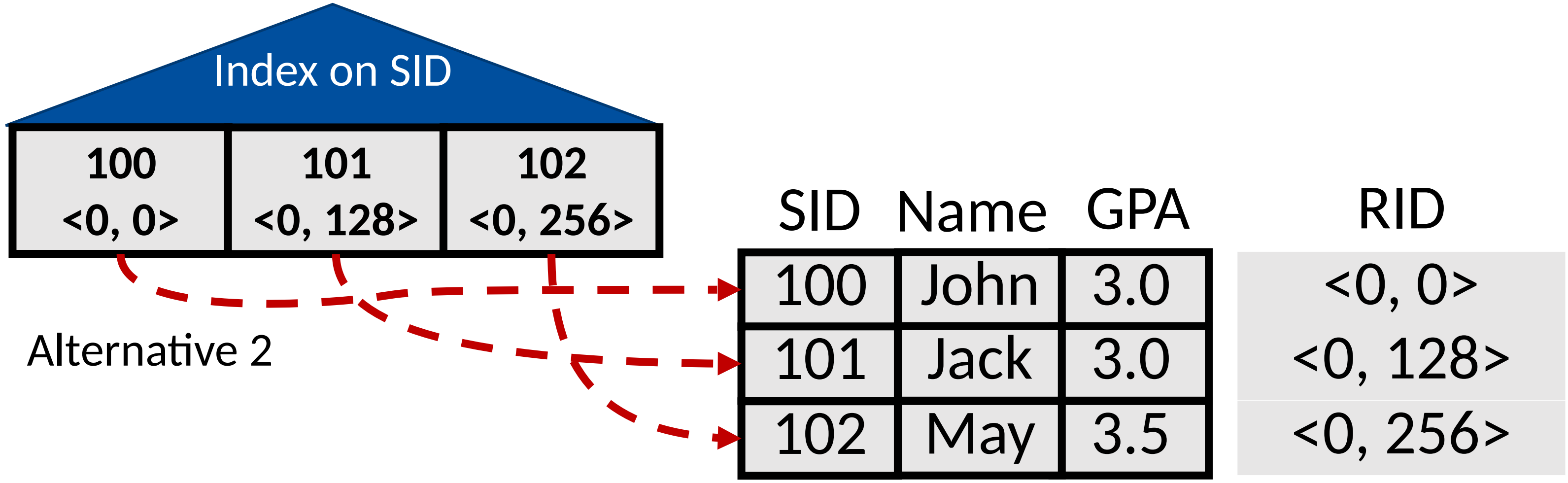
- A special file organization, index == file
- At most one such index per table
  - May need to duplicate data otherwise

Alternative 2: <key, RID>

- Map keys to RIDs
- Independent of the table's organization



Alternative 1



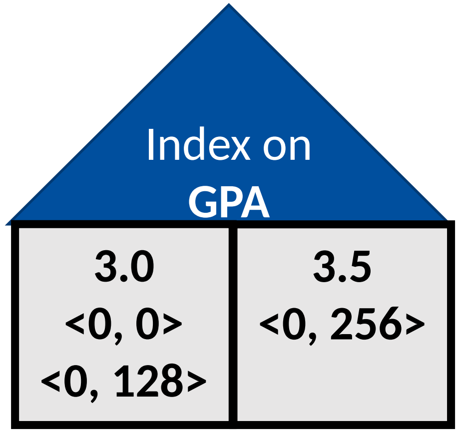
Alternative 2

Recap: Index Data Entries

Alternative 3: <key, RID list>

- A list of records that match the search key
- Independent of the table's organization

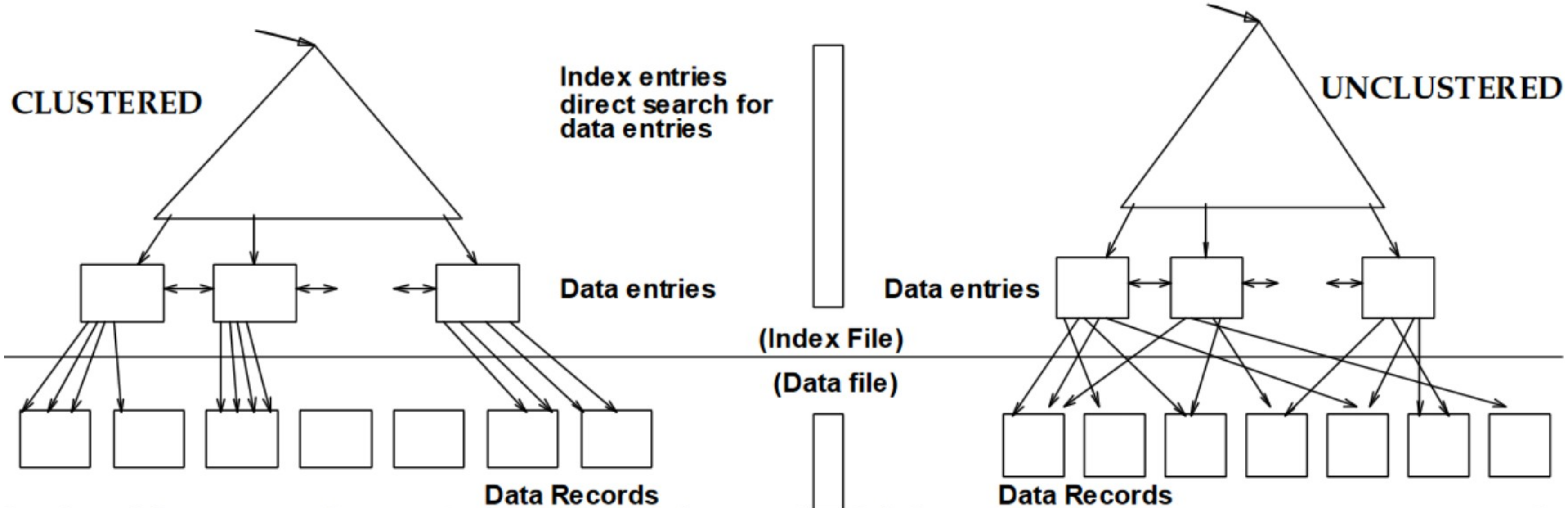
	Name	GPA	RID
100	John	3.0	<0, 0>
101	Jack	3.0	<0, 128>
102	May	3.5	<0, 256>



Alternatives 2 vs. 3:

- Basically the same for unique indexes
- For non-unique indexes, Alternative 3 provides better space utilization (no repeated key storage)

Recap: Clustered vs. Unclustered Index



Order in index == order in data file

- Better performance for scans (faster sequential reads)

Order in index != order in data file

- May be slower: potentially more random reads

Alternative 1 (key → actual record): always clustered

Alternative 2 and 3 (key → RIDs): could be clustered or non-clustered

Index Nested Loop Join

Cost: M + pR * M * cost to find matching tuples in S (using index)

Case 1: index stores records directly as data entries (Alternative 1)

- Cost to traverse from root to leaf in trees or loading hash table pages

Case 2: index stores RIDs or RID lists (Alternatives 2 and 3)

- Cost = cost to look up RIDs + cost to retrieve actual records
- Look up RIDs
  - E.g., traverse B-tree, typically 2-4 I/Os
- Retrieving records from RIDs
  - Unclustered index: up to 1 I/O per matching S tuple
  - Clustered index: 1 I/O per page of matching S tuples
    - Cannot apply the equation: “M + pR * M * cost to find matching tuples in S”

Example: Index Nested Loop Join

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

Reserves table:
- Tuple size: 40 bytes
- 100 tuples per page
- 1000 pages in total

Sailors table:
- Tuple size: 50 bytes
- 80 tuples per page
- 500 pages in total

Query: Join Sailors and Reserves on sid column

Cost: M + pR * M * cost to find matching tuples in S (using index)

- Sailors as the inner table, with index on Sailors.sid
  - M = 1000 (# of pages in Reserves), pR = 100 * 1000
- Cost to find matching tuples using index (assuming a B-tree with 3 levels including internal and leaf levels)
  - With an unclustered B-tree index: 3 I/Os for tree traversal + 1 I/O for accessing heap file

    ➔ Cost = M + pR * M * 4 = 1000 + 100 * 1000 * 4
  - With a clustered B-tree index: 3 I/O for tree traversal, but heap access is based on the number of distinct values of the outer table (1 I/O per distinct value)

    ➔ Cost = M + pR * M * 3 + distinct_vals(Reservers) = 1000 + 100 * 1000 * 3 + X where X <= max number of tuples in Reserves (100 * 1000)

Sort-Merge Join

Two stages:

- Sort: sort tuples in relations R and S by the join key
- Join: merge-scan the sorted relations, output matching tuples

Example query: Reserves $\bowtie_{\text{Reserves.sid = Sailors.sid}}$ Sailors

Reserves (sorted)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

Sailors (sorted)

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

Sort-Merge Join

- Keep two cursors (r, s) pointing to the "current" Reserves tuple and Sailors tuple
- Advance r until *r >= *s, advance s until *s >= *r
- Mark the start of the current "block" in S, i.e., let mark = s and start to generate output tuples

➔ There might be repeated values in R, the mark allows us to "come back" to join the matching Sailors tuples for each duplicate R tuple

Step 0:

Reserves (sorted)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

Sailors (sorted)

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

r

s

m

Sort-Merge Join

Step 1: **Reserves** (sorted)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

Sailors (sorted)

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

r

s

m

- Advanced both r and s, mark s
- Current *r matches *s: output result and advance s

Sort-Merge Join

Step 2: **Reserves** (sorted)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

r

Sailors (sorted)

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

s

m

Output (so far):

sid	bid	day	rname	sname	rating	age
28	103	Jan 3, 2019	Guppy	Yuppy	9	35

Now *r and *s do not match, next step:

- Reset s to m, advance r, reset mark to null

Sort-Merge Join

Step 3: **Reserves** (sorted)

r →

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

Sailors (sorted)

s →

|| ← m

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

Output (so far):

sid	bid	day	rname	sname	rating	age
28	103	Jan 3, 2019	Guppy	Yuppy	9	35

Next:

- Position r and s so that for Reserves *r >= *s, for Sailors *s >= *r, and mark s

Sort-Merge Join

Step 4: **Reserves** (sorted)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

r

Sailors (sorted)

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

s

m

Output (so far):

sid	bid	day	rname	sname	rating	age
28	103	Jan 3, 2019	Guppy	Yuppy	9	35

Current *r matches *s, next:

- Output result and advance s

Sort-Merge Join

Step 5: **Reserves** (sorted)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

Sailors (sorted)

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

r s m

Output (so far):

sid	bid	day	rname	sname	rating	age
28	103	Jan 3, 2019	Guppy	Yuppy	9	35
28	103	Jan 4, 2019	Yuppy	Yuppy	9	35

Now *r and *s do not match, next step:

- Reset s to m, advance r, reset mark to null

Sort-Merge Join

Step 6: **Reserves** (sorted) — r → 31 (Mar 3, 2019)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

Sailors (sorted) — s → 28; || ← m

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
31	Lubber	8	55
44	Guppy	5	35
58	Rusty	10	35

Output (so far):

sid	bid	day	rname	sname	rating	age
28	103	Jan 3, 2019	Guppy	Yuppy	9	35
28	103	Jan 4, 2019	Yuppy	Yuppy	9	35

Next:

- Position r and s so that for Reserves *r >= *s, for Sailors *s >= *r, and mark s

Sort-Merge Join

Step 7: **Reserves** (sorted)

sid	bid	day	rname
28	103	Jan 3, 2019	Guppy
28	103	Jan 4, 2019	Yuppy
r → 31	101	Mar 3, 2019	Dustin
31	102	Apr 25, 2019	Lubber
31	101	May 12, 2019	Lubber
58	103	May 20, 2019	Dustin

Sailors (sorted)

sid	sname	rating	age
22	Dustin	7	45
28	Yuppy	9	35
s → 31	Lubber	8	55 ← m
44	Guppy	5	35
58	Rusty	10	35

Output (so far):

sid	bid	day	rname	sname	rating	age
28	103	Jan 3, 2019	Guppy	Yuppy	9	35
28	103	Jan 4, 2019	Yuppy	Yuppy	9	35

Repeat the previous process until r is EOF

Sort-Merge Join Algorithm

```
Advance r until *r >= *s, advance s until *s >= *r
while (*r != EOF) {
  if (m == null) {
    while (*r < *s) { advance r; }
    while (*s < *r) { advance s; }
    m = s
  }
  if (*r == *s) {
    output <*r, *s>
    advance s
  } else {
    s = m;
    advance r
    m = null
  }
}
```

Mark the start of the potential block of matching records in S

*r and *s match:

- Output the join result
- Advance to the next S tuple to find more matches for the same R tuple (*r)

*r and *s do not match

- Advance to the next R tuple and reset to the marked S tuple (*m)
- Need to reset m to null to allow advancing r and s if needed in the next round

Sort-Merge Join (R ⋈ S)

- R is scanned once
- S could be scanned multiple times
  - Each ‘block’ is scanned per matching R tuple
  - But usually the needed pages are already in buffer pool

Cost: sorting cost for R and S + scanning cost

- Scanning cost
  - Best scenario: M + N, if no S block is scanned multiple times
  - Worst scenario: M * N, if the repeatedly scanned S block cannot be buffered
    - E.g., the page is no longer in buffer pool when it is requested the second time
    - Query optimizer tries to avoid this when deciding on plans

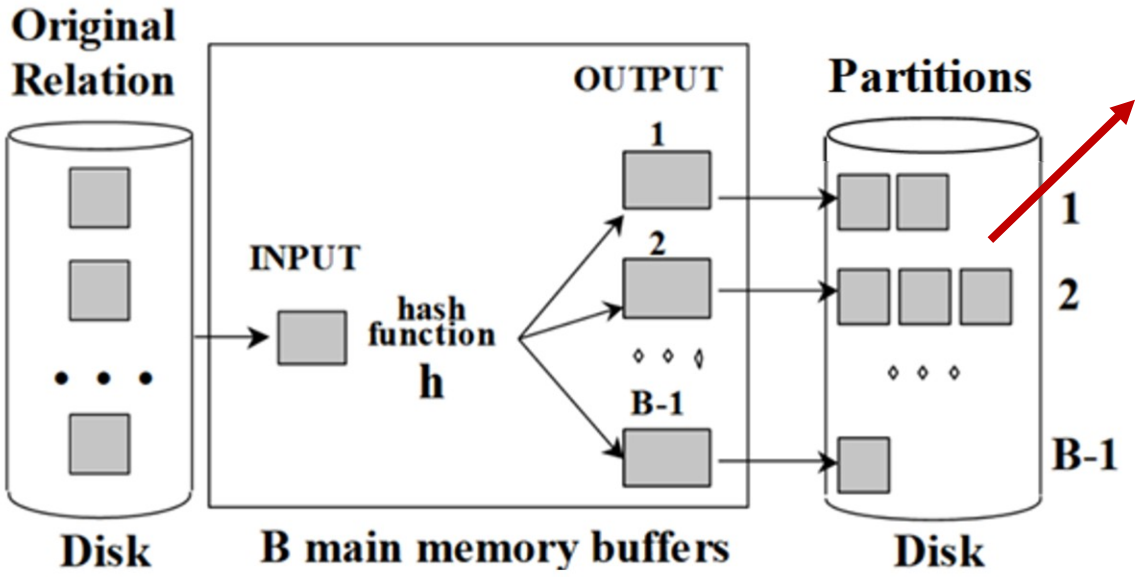
(Grace) Hash Join (R ⋈ S)

Leverage hashing to partition input relations, then join by partitions

- Two phases: building and probing

Building (aka partitioning) phase:

- Hash both relations on the join attribute using the same hash function **h** into k partitions
- Partitions could be on different disks or even machines

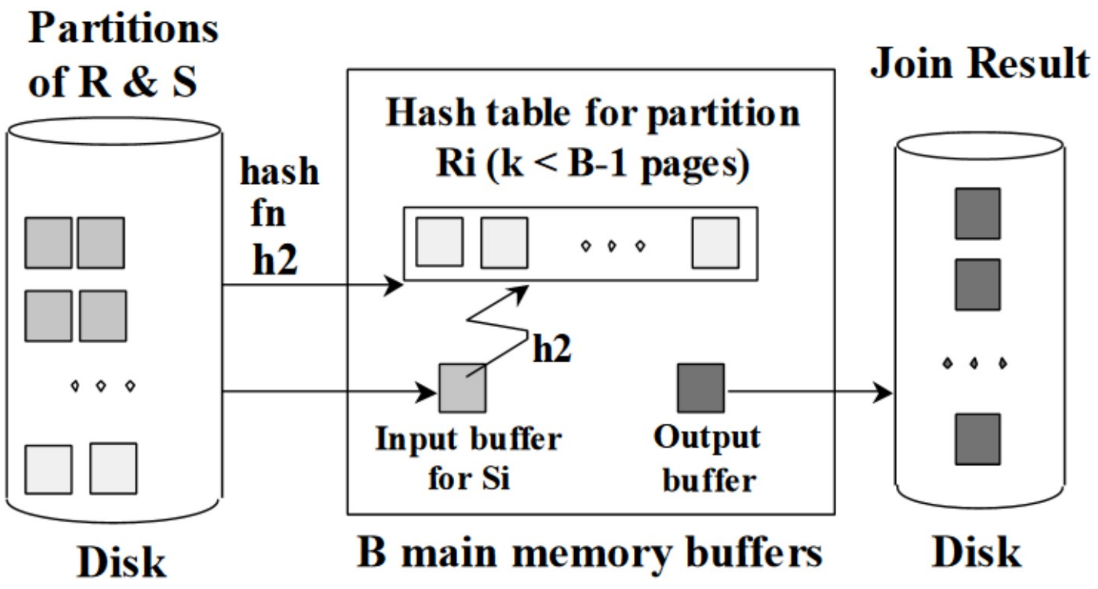


Desirable to have each partition fit in memory

(Grace) Hash Join (R ⋈ S)

Probing (aka matching) phase:

- After the building phase, R tuples in partition i can join only with S tuples in its partition i
- Read each R partition and scan just the corresponding S partition for matches
- Reduce CPU cost on matching:
  - Build a hash table for the R partition with a different hash function **h2**
  - Scan S partition and probe the hash table using h2 to find matches



Cost of Grace Hash Join (R ⋈ S)

Suppose R has M pages and S has N pages

Number of I/O operations:

- Partitioning phase
  - Read and write both relations: 2 * (M + N) I/Os
- Matching phase
  - Read both relations once: (M + N) I/Os
- Total: 3 * (M + N) I/Os

Sort-Merge Join vs. Hash Join

Sort-merge join:

- Good if the input is already sorted or the output needs to be sorted
- Less sensitive to data skew or bad hash functions

Hash join:

- Good if the input is already hashed or the output needs to be hashed
- May be vulnerable to data skew/bad hash functions
- Number of passes depends on the size of the smaller relation

General Join Conditions

Equalities over multiple attributes:

- E.g., R.sid = S.sid AND R.rname = S.sname
- Index nested loops join
  - Build or use index on <sid, sname> (S being the inner relation)
- Sort-merge/hash join
  - Sort/partition on the combined attributes

Inequality conditions:

- E.g., R.age < S.age
- Cannot use hash join or sort-merge join
- Index nested loops join: need index that supports range scans
  - Range probes the inner relation
  - Number of matches likely much higher than equality join

Impact of Buffering

- If several operations are executing concurrently, estimating the available and allocating buffer pages is guesswork
- **Repeated access pattern interacts with buffer replacement policy**
- Example 1: the inner relation is often scanned repeatedly in tuple-at-a-time nested loops join
  - LRU could be the worst algorithm to use (sequential flooding)
  - MRU would be much better
- Example 2: in block nested loops join, B – 2 buffers are used for the outer relation, 1 for the result
  - The remaining one buffer for reading the inner relation
  - Replacement policy does not matter here

Summary

- Join algorithms
  - Nested loops (tuple/page-at-a-time, block, index)
  - Sort-merge join and hash join
  - Costs, pros and cons
- Common techniques
  - Sorting, hashing, partitioning
  - Index can be helpful and preferred in general to reduce cost
  - Sometimes also need to scan the entire relation
- Buffering impact
  - Relations are sometimes repeatedly scanned
  - Need to watch out for problems like sequential flooding in LRU