

Evaluation of Relational Operations

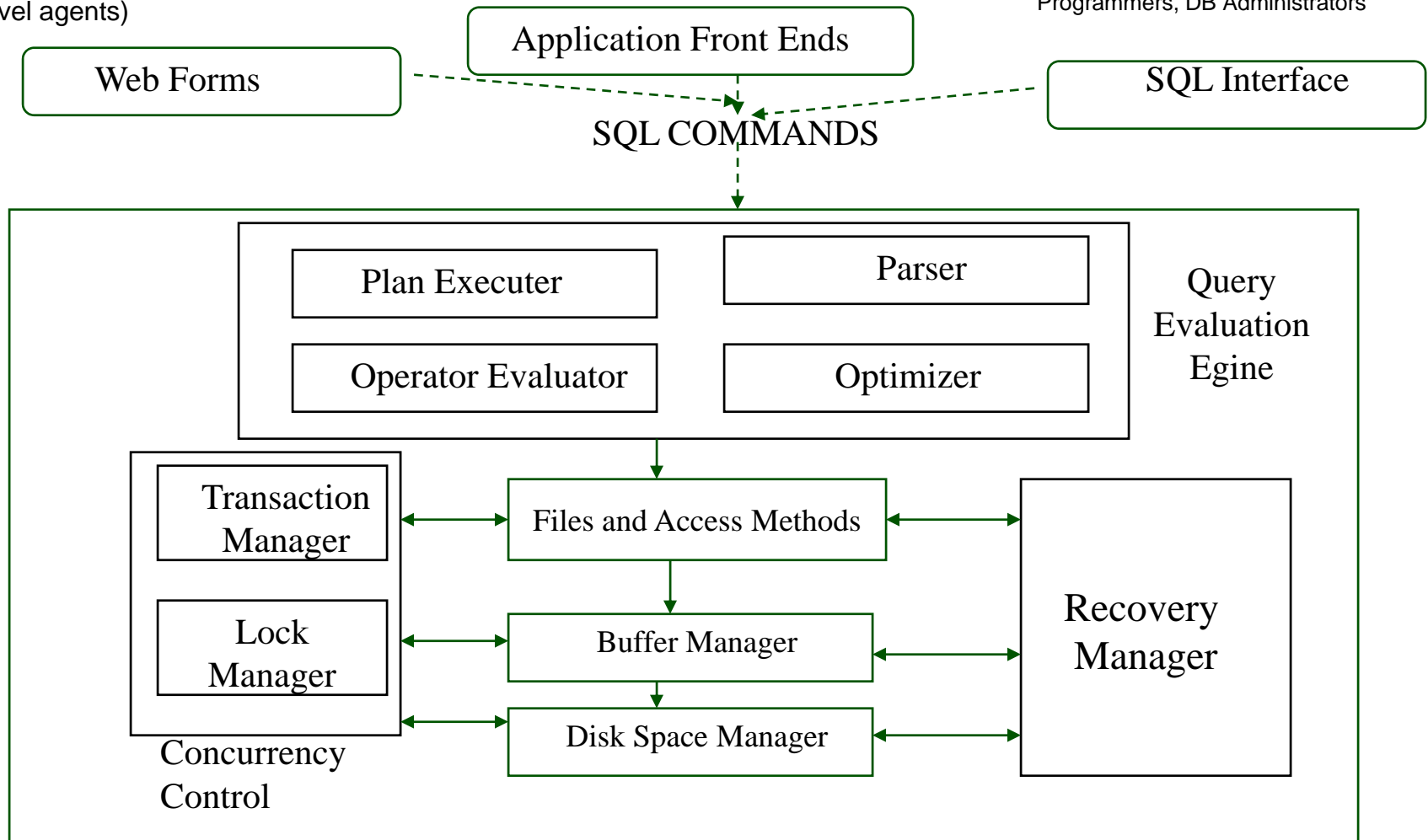
Chapter 14

Relational Operations

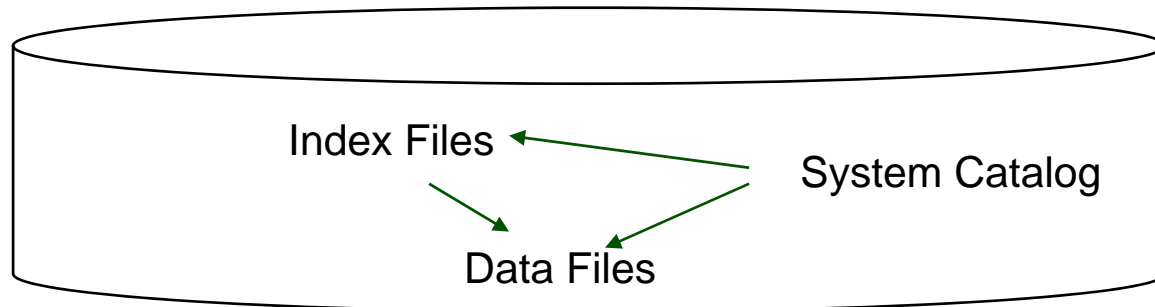
- ❖ We will consider how to implement:
 - Selection (σ) Selects a subset of rows from relation.
 - Projection (π) Deletes unwanted columns from relation.
 - Join (\bowtie) Allows us to combine two relations.
 - Set-difference ($-$) Tuples in reln. 1, but not in reln. 2.
 - Union (\cup) Tuples in reln. 1 and in reln. 2.
 - Aggregation (SUM, MIN, etc.) and GROUP BY
- ❖ Since each op returns a relation, ops can be *composed*!
After we cover the operations, we will discuss how to *optimize* queries formed by composing them.

Unsophisticated users (Customers, Travel agents)

Sophisticated users, application Programmers, DB Administrators



Architecture Of DBMS



Schema for Examples

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

- ❖ Similar to old schema; *rname* added for variations.
- ❖ Reserves:
 - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages.
- ❖ Sailors:
 - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Outline

- ❖ The Selection Operation
- ❖ The Projection operation
- ❖ Join Operation

The Selection Operation

- ❖ Sample selection query

```
SELECT *
```

```
FROM Reserves R
```

```
WHERE R.rname='Joe'
```

- ❖ We can evaluate the query by scanning entire relation and checking the condition on each tuple
 - Cost is 1000 pages (Reserves contains 1000 pages)
- ❖ If few tuples have “rname=Joe”, this is expensive.
- ❖ We can exploit B+tree on Joe, but index on *bid* would not be useful.

Selection

- ❖ No index, Unsorted data
 - Scan entire relation, so selective access path is scan.
- ❖ No index, Sorted data
 - Do binary search
 - Access path is sorted file scan
- ❖ B+tree
 - Clustered B+tree
 - If op is not equality, use index.
 - If op is equality, hash index is better, if it exists
 - Unclustered B+tree
 - Cost of using the index depends on the number of tuples that satisfy the selection.
 - Use of unclustered for range selection is expensive, it is better to scan the relation.

Using an Index for Selections

- ❖ Cost depends on #qualifying tuples, and clustering.
 - Cost of finding qualifying data entries (typically small) plus cost of retrieving records (could be large w/o clustering).
 - In example, assuming uniform distribution of names, about 10% of tuples qualify (100 pages, 10000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10000 I/Os!
- ❖ *Important refinement for unclustered indexes:*
 1. Find qualifying data entries.
 2. Sort the rid's of the data records to be retrieved.
 3. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

The Selection

- ❖ Follow Hash Index for Equality condition

Two Approaches to General Selections

- ❖ First approach: Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't **match** the index:
 - *Most selective access path*: An index or file scan that we estimate will require the fewest page I/Os.
 - Terms that match this index reduce the number of tuples *retrieved*; other terms are used to discard some retrieved tuples, but do not affect number of tuples/pages fetched.
 - Consider *day<8/9/94 AND bid=5 AND sid=3*.
 - A B+ tree index on *day* can be used; then, *bid=5* and *sid=3* must be checked for each retrieved tuple. Similarly, a hash index on *<bid, sid>* could be used; *day<8/9/94* must then be checked.

Intersection of Rids

- ❖ Second approach (if we have 2 or more matching indexes):
 - Get sets of rids of data records using each matching index.
 - Then *intersect* these **sets of rids** (we'll discuss intersection soon!)
 - Retrieve the records and apply any remaining terms.
 - Consider *day<8/9/94 AND bid=5 AND sid=3*. If we have a B+ tree index on *day* and an index on *sid*, we can retrieve rids of records satisfying *day<8/9/94* using the first, rids of recs satisfying *sid=3* using the second, intersect, retrieve records and check *bid=5*.

The Projection Operation

```
SELECT  DISTINCT  
        R.sid, R.bid  
FROM    Reserves R
```

- ❖ Remove unwanted attributes
- ❖ Eliminate duplicates

The Projection Based on Sorting

- ❖ **Modify Pass 0 of external sort to eliminate unwanted fields.**

Thus, runs of about 2B pages are produced, but tuples in runs are smaller than input tuples. (Size ratio depends on # and size of fields that are dropped.)

- ❖ **Modify merging passes to eliminate duplicates.**

- Thus, number of result tuples smaller than input. (Difference depends on # of duplicates.)

- ❖ **Cost:**

- In Pass 0, read original relation (size M), write out same number of smaller tuples. In merging passes, fewer tuples written out in each pass. Using Reserves example, 1000 input pages reduced to 250 in Pass 0 if size ratio is 0.25

Projection Based on Hashing

- ❖ *Partitioning phase*: Read R using one input buffer. For each tuple, discard unwanted fields, apply hash function $h1$ to choose one of B-1 output buffers.
 - Result is B-1 partitions (of tuples with no unwanted fields).
2 tuples from different partitions guaranteed to be distinct.
- ❖ *Duplicate elimination phase*: For each partition, read it and build an in-memory hash table, using hash fn $h2$ ($\neq h1$) on all fields, while discarding duplicates.
 - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.
- ❖ *Cost*: For partitioning, read R, write out each tuple, but with fewer fields. This is read in next phase.

Discussion of Projection

- ❖ Sort-based approach is the standard; better handling of skew and result is sorted.
- ❖ If an index on the relation contains all wanted attributes in its search key, can do *index-only* scan.
 - Apply projection techniques to data entries (much smaller!)
- ❖ If an ordered (i.e., tree) index contains all wanted attributes as *prefix* of search key, can do even better:
 - Retrieve data entries in order (index-only scan), discard unwanted fields, compare adjacent tuples to check for duplicates.

Equality Joins With One Join Column

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

- ❖ In algebra: $R \bowtie S$. Common! Must be carefully optimized. $R \times S$ is large; so, $R \times S$ followed by a selection is inefficient.
- ❖ Assume: M tuples in R , p_R tuples per page, N tuples in S , p_S tuples per page.
 - In our examples, R is Reserves and S is Sailors.
- ❖ We will consider more complex join conditions later.
- ❖ *Cost metric*: # of I/Os. We will ignore output costs.

Simple Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S do
        if  $r_i == s_j$  then add  $\langle r, s \rangle$  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 $\langle r, s \rangle$, 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 = $500 + 500 * 1000$

Index Nested Loops Join

```
foreach tuple r in R do
    foreach tuple s in S where  $r_i == s_j$  do
        add  $\langle r, s \rangle$  to result
```

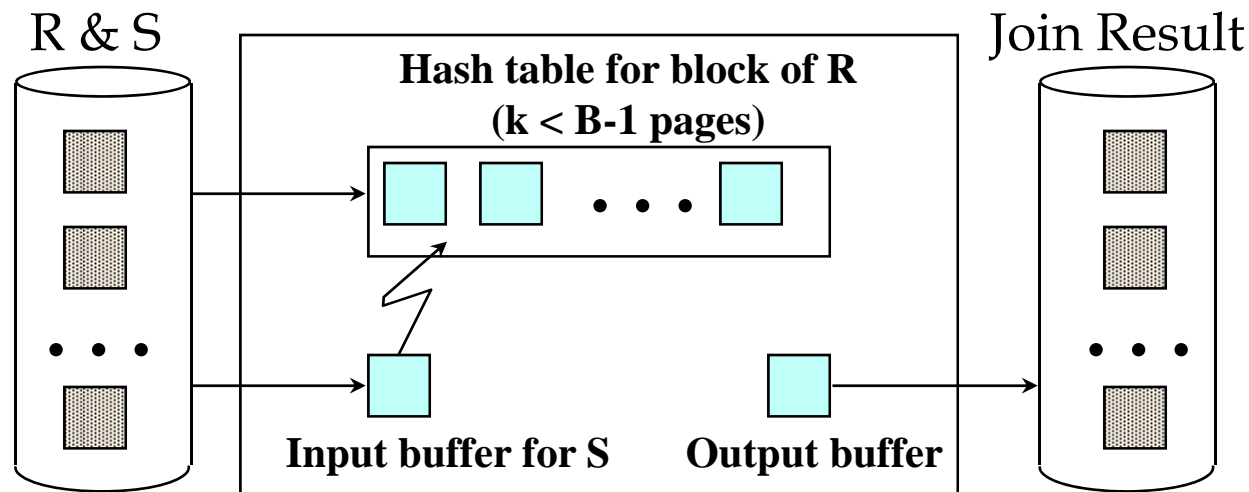
- ❖ If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
 - Cost: $M + (M * p_R) * \text{cost of finding matching S tuples}$
- ❖ For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples depends on clustering.
 - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

Examples of Index Nested Loops

- ❖ Hash-index on *sid* of Sailors (as inner):
 - Scan Reserves: 1000 page I/Os, 100×1000 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.
- ❖ Hash-index on *sid* of Reserves (as inner):
 - Scan Sailors: 500 page I/Os, 80×500 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 whether the index is clustered.

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 ``block'' of outer R.
 - For each matching tuple r in R-block, s in S-page, add $\langle r, s \rangle$ to result. Then read next R-block, scan S, etc.



Examples of Block Nested Loops

- ❖ **Cost: Scan of outer + #outer blocks * scan of inner**
 - $\text{\#outer blocks} = \lceil \text{\# of pages of outer} / \text{blocksize} \rceil$
- ❖ With Reserves (R) as outer, and 100 pages of R:
 - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
 - Per block of R, we scan Sailors (S); 10*500 I/Os.
 - If space for just 90 pages of R, we would scan S 12 times.
- ❖ With 100-page block 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.
- ❖ With sequential reads considered, analysis changes:
may be best to divide buffers evenly between R and S.

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 col.), and output result tuples.
 - Advance scan of R until current R-tuple \geq current S tuple, then advance scan of S until current S-tuple \geq current R tuple; do this until current R tuple = current S tuple.
 - At this point, all R tuples with same value in R_i (*current R group*) and all S tuples with same value in S_j (*current S group*) match; output $\langle r, s \rangle$ for all pairs of such tuples.
 - Then resume scanning R and S.
- ❖ R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

Example of Sort-Merge Join

<u>sid</u>	sname	rating	age	<u>sid</u>	<u>bid</u>	<u>day</u>	rname
22	dustin	7	45.0	28	103	12/4/96	guppy
28	yuppy	9	35.0	28	103	11/3/96	yuppy
31	lubber	8	55.5	31	101	10/10/96	dustin
44	guppy	5	35.0	31	102	10/12/96	lubber
58	rusty	10	35.0	31	101	10/11/96	lubber
				58	103	11/12/96	dustin

❖ **Cost:** $M \log M + N \log N + (M+N)$

- The cost of scanning, $M+N$, could be $M*N$ (very unlikely!)

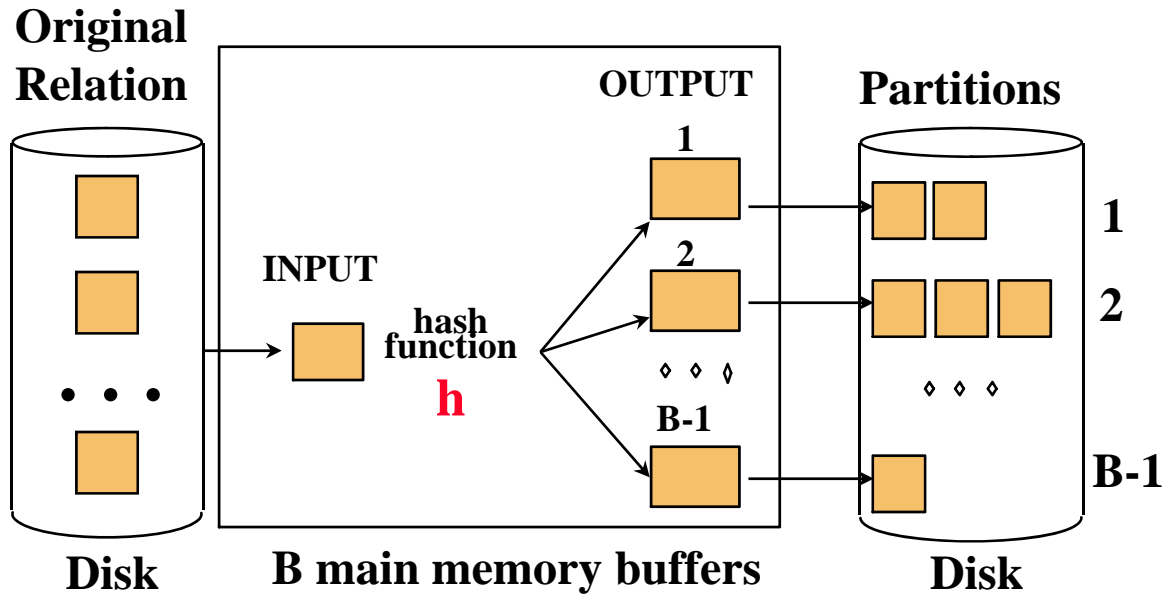
❖ With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500.
(BNL cost: 2500 to 15000 I/Os)

Refinement of Sort-Merge Join

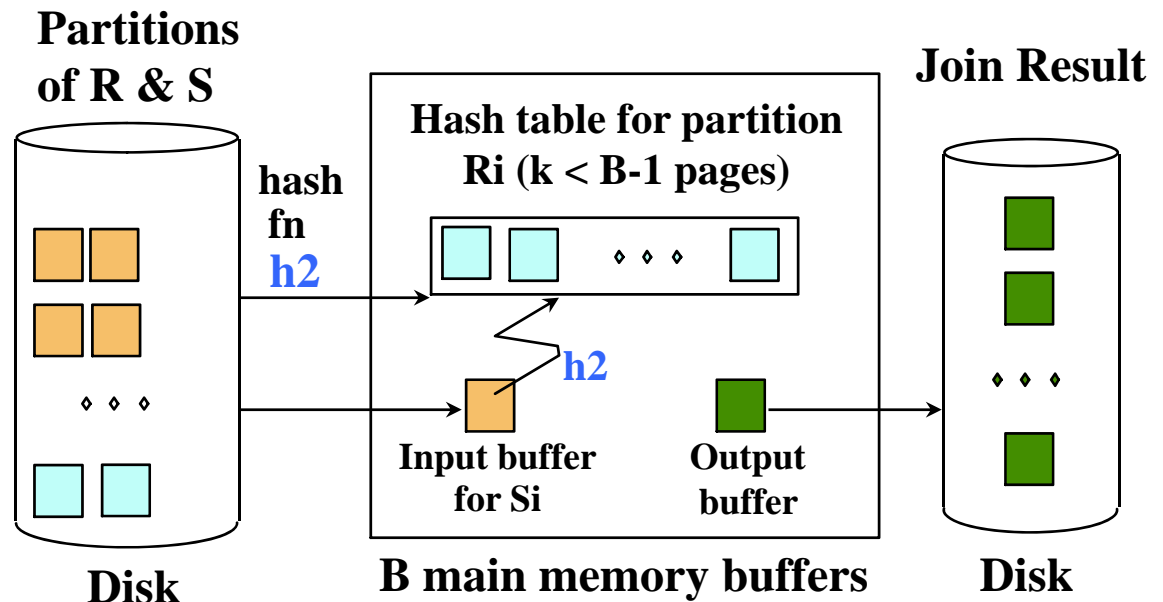
- ❖ We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
 - With $B > \sqrt{L}$, where L is the size of the larger relation, using the sorting refinement that produces runs of length $2B$ in Pass 0, #runs of each relation is $< B/2$.
 - Allocate 1 page per run of each relation, and 'merge' while checking the join condition.
 - **Cost:** read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
 - In example, cost goes down from 7500 to 4500 I/Os.
- ❖ In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.

Hash-Join

❖ Partition both relations using hash fn **h**: R tuples in partition *i* will only match S tuples in partition *i*.



❖ Read in a partition of R, hash it using **h2** (\neq **h**!). Scan matching partition of S, search for matches.



Observations on Hash-Join

- ❖ #partitions $k < B-1$ (why?), and $B-2 > \text{size of largest partition}$ 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 apply hash-join technique recursively to do the join of this R -partition with corresponding S -partition.

Cost of 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 (*what is this, for each?*) both have a cost of $3(M+N)$ I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
 - Sort-Merge less sensitive to data skew; result is sorted.

General Join Conditions

- ❖ 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/partition on combination of the two join columns.
- ❖ Inequality conditions (e.g., *R.rname < S.sname*):
 - For Index NL, need (clustered!) B+ tree index.
 - Range probes on inner; # matches likely to be much higher than for equality joins.
 - Hash Join, Sort Merge Join not applicable.
 - Block NL quite likely to be the best join method here.

Set Operations ($R \cap S$, $R \times S$, $R - S$, $R \cup S$)

- ❖ Intersection and cross-product special cases of join.
- ❖ Union (Distinct) and Except similar; we'll do union.
- ❖ Sorting based approach to union:
 - Sort both relations (on combination of all attributes).
 - Scan sorted relations and merge them.
 - *Alternative:* Merge runs from Pass 0 for *both* relations.
- ❖ Hash based approach to union:
 - Partition R and S using hash function h .
 - For each S-partition, build in-memory hash table (using h_2), scan corr. R-partition and add tuples to table while discarding duplicates.

Aggregate Operations (AVG, MIN, etc.)

❖ Without grouping:

- In general, requires scanning the relation.
- Given index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.

❖ With grouping:

- Sort on group-by attributes, then scan relation and compute aggregate for each group. (Can improve upon this by combining sorting and aggregate computation.)
- Similar approach based on hashing on group-by attributes.
- Given tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan; if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

Impact of Buffering

- ❖ If several operations are executing concurrently, estimating the number of available buffer pages is guesswork.
- ❖ Repeated access patterns interact with buffer replacement policy.
 - e.g., Inner relation is scanned repeatedly in Simple Nested Loop Join. With enough buffer pages to hold inner, replacement policy does not matter. Otherwise, MRU is best, LRU is worst (*sequential flooding*).
 - Does replacement policy matter for Block Nested Loops?
 - What about Index Nested Loops? Sort-Merge Join?

Summary

- ❖ A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned (and it is important to do this!).
- ❖ Many alternative implementation techniques for each operator; no universally superior technique for most operators.
- ❖ Must consider available alternatives for each operation in a query and choose best one based on system statistics, etc. This is part of the broader task of optimizing a query composed of several ops.