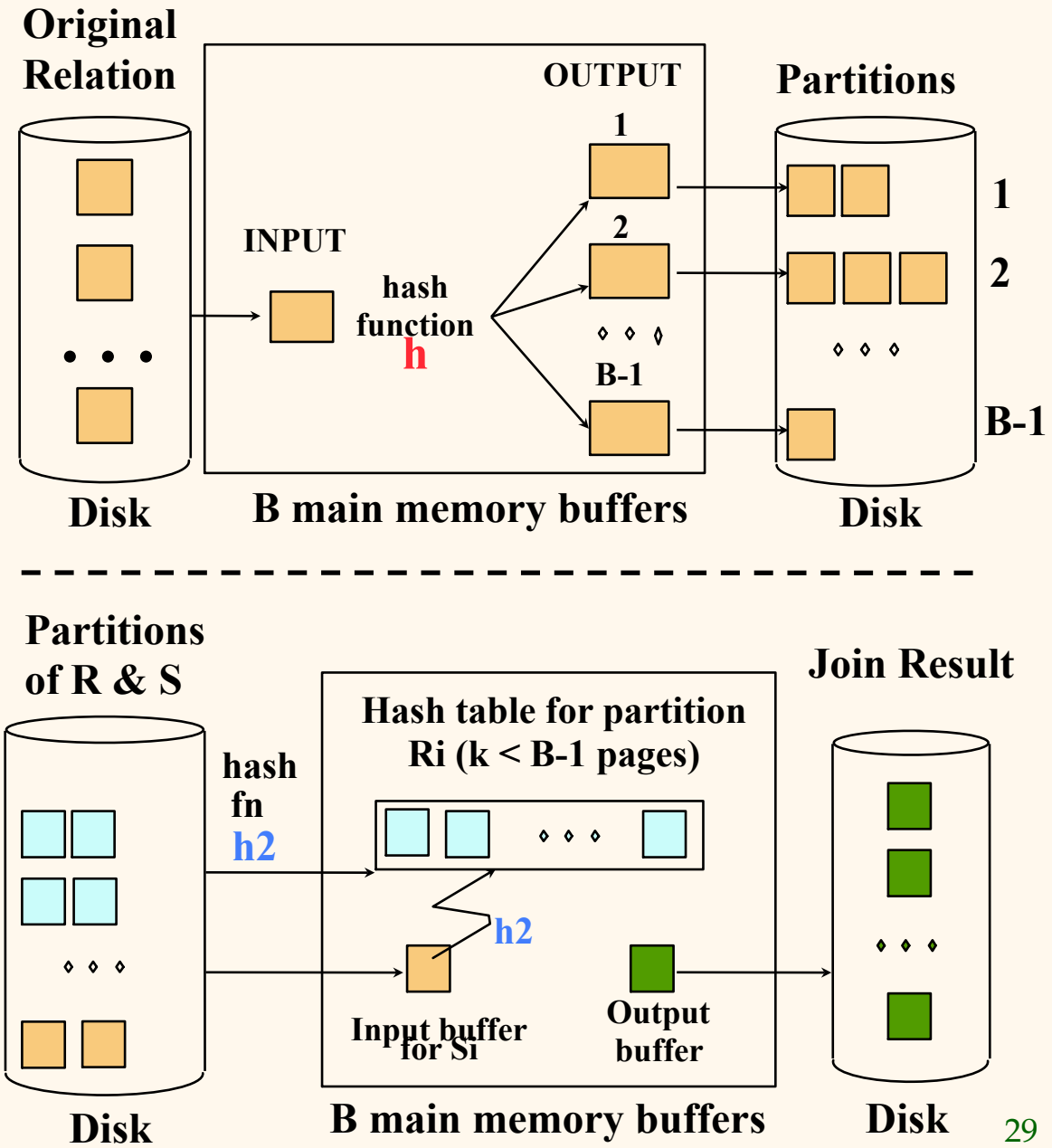
Hash-Join

- Partitioning: Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.
- Probing: Read in partition i of R, build hash table on Ri using **h2 (<> h!)**. Scan partition i of S, search for matches.



Observations on Hash-Join

- # partitions ≤ B-1, and size of largest partition ≤ B-2 to be held in memory. Assuming uniformly sized partitions, we get:
  - M / (B-1) < (B-2), i.e., B must be $>\sqrt{M}$
  - Hash-join works if the smaller relation satisfies above.
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If hash function h 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

- Partitioning reads+writes both relns; 2(M+N). Probing reads both relns; M+N I/Os. The total is 3(M+N).
  - In our running example, a total of 4500 I/Os using hash join, less than 1 min (compared to 140 hours w. NLJ).

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* and check the other join condition on the fly.
  - 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 B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins (clustered index is much preferred).
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be a winner here.

Outline

- Sorting
- Evaluation of joins
- Evaluation of other operations

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).
  - Consider a selection of the form *gpa* > 3.0 and assume 10% of tuples qualify (100 pages, 10,000 tuples). With a clustered index, cost is little more than 100 I/Os; if unclustered, upto 10,000 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.

Two Approaches to General Selections

- First approach: (1) Find the *most selective access path*, *retrieve* tuples using it, and (2) apply any remaining terms that don't match the index *on the fly*.
  - *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.
    - A hash index on <*bid, sid*> could be used; *day<8/9/94* must then be checked on the fly.

Intersection of Rids

- Second approach (if we have 2 or more matching indexes that use Alternatives (2) or (3) for data entries):
  - Get sets of rids of data records using each matching index.
  - Then *intersect* these sets of rids.
  - 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*, both using Alternative (2), we can:
    - retrieve rids of records satisfying *day<8/9/94* using the first, rids of records satisfying *sid=3* using the second,
    - intersect these rids,
    - retrieve records and check *bid=5*.

The Projection Operation

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

- Projection consists of two steps:
  - Remove unwanted attributes (i.e., those not specified in the projection).
  - Eliminate any duplicate tuples that are produced.
- Algorithms: single relation sorting and hashing based on all remaining attributes.

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* (<> *h1*) on all fields, while discarding duplicates.
  - If partition does not fit in memory, can apply hash-based projection algorithm recursively to this partition.

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.

Set Operations

- Intersection and cross-product special cases of join.
  - Intersection: equality on all fields.
- 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, removing duplicates.
- Hash based approach to union:
  - Partition R and S using hash function *h*.
  - For each S-partition, build in-memory hash table (using *h2*). Scan R-partition. For each tuple, probe the hash table. If the tuple is in the hash table, discard it; o.w. add it to the hash table. Write out hash table and clear it for next partition.

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 (GROUP BY):
  - Sort on group-by attributes, then scan relation and compute aggregate for each group.
  - 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.

Summary

- A virtue of relational DBMSs: *queries are composed of a few basic operators*; the implementation of these operators can be carefully tuned.
- Algorithms for evaluating relational operators use some simple ideas extensively:
  - Indexing: Can use WHERE conditions to retrieve small set of tuples (selections, joins)
  - Iteration: Sometimes, faster to scan all tuples even if there is an index. (And sometimes, we can scan the data entries in an index instead of the table itself.)
  - Partitioning: By using sorting or hashing, we can partition the input tuples and replace an expensive operation by similar operations on smaller inputs.

Summary (contd)

- Many 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 state (e.g., memory) and
  - statistics (table size, # tuples matching value k).
- This is part of the broader task of optimizing a query composed of several operations.