External Sort

Kathleen Durant PhD
Lecture 18 CS 3200
Northeastern University

Outline for today

- External Sort
- Review of Sort-Merge Join Algorithm
- Refinement: 2 Pass Sort Merge Join Algorithm
- Algorithms for other RA operators

Why Sort?

- A classic problem in computer science
- A precursor to other algorithms like search and merge
- Important utility in DBMS:
  - Data requested in sorted order (e.g., ORDER BY)
    - e.g., find students in increasing *gpa* order
  - Sorting useful for eliminating *duplicate copies* in a collection of records (e.g., SELECT DISTINCT)
  - *Sort-merge* join algorithm involves sorting.
  - Sorting is first step in *bulk loading* B+ tree index.

Problem: sort 1TB of data with 1GB of RAM. Key is to minimize # I/Os

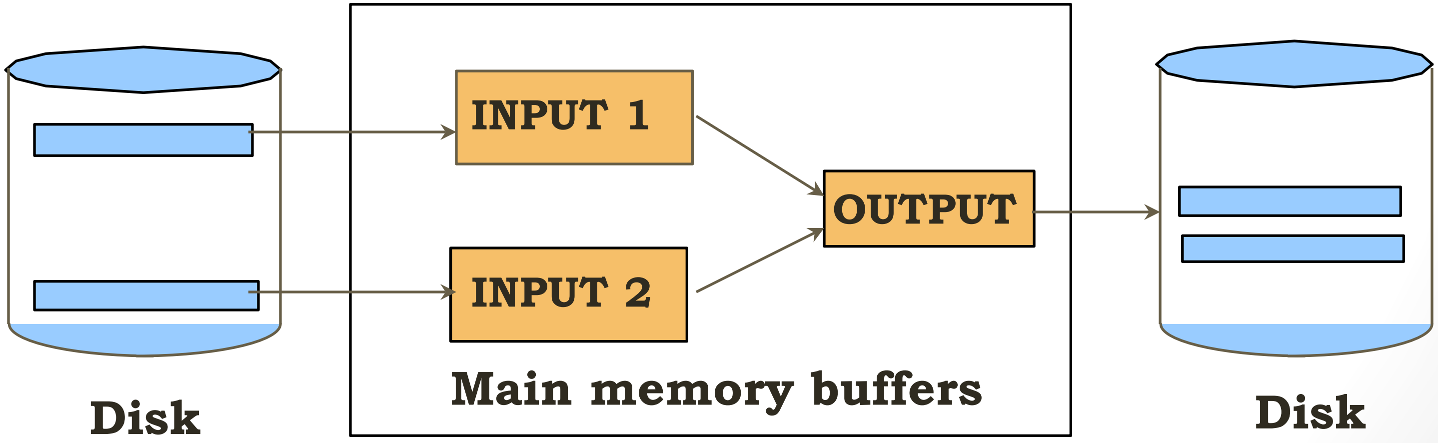
External Sorts

- Two-Way Merge Sort
  - Simplified case (pedagogical)
- General External Merge Sort
  - Takes better advantage of available memory
  - Performance Optimizations
  - Blocked I/O
  - Double Buffering
- Replacement Sort
- Using B+ trees for Sort

2-Way Sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.

Partition data
Pass determines
Size of partition



Two-Way External Merge Sort

- *Divide and conquer*, sort subfiles (runs) and merge

A file of N pages:

Pass 0: N sorted runs of 1 page each

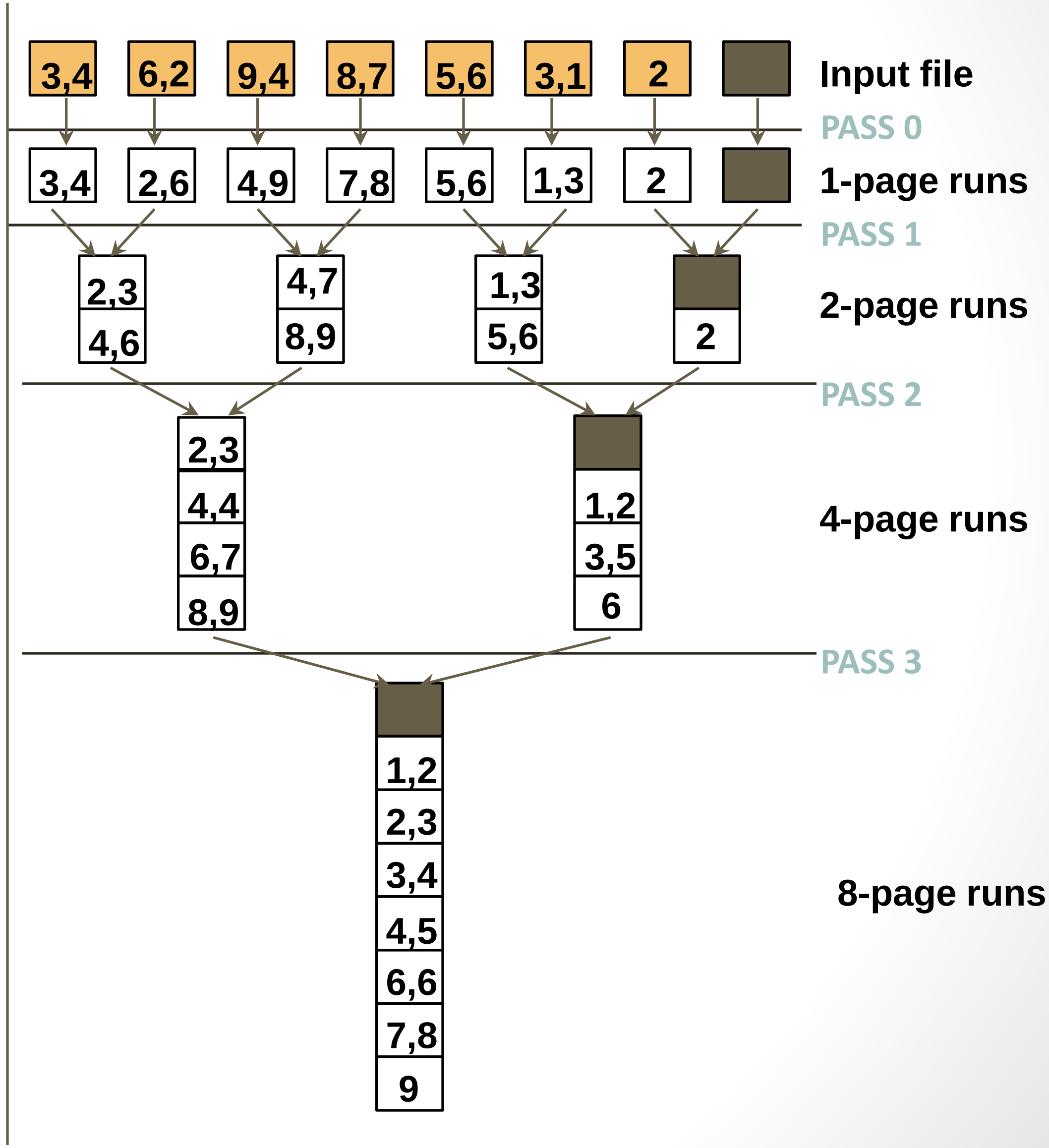
Pass 1: N/2 sorted runs of 2 pages each

Pass 2: N/4 sorted runs of 4 pages each

…

Pass P: 1 sorted run of 2^P pages

$2^P \geq N \rightarrow P \geq \log_2 N$



Cost: Two-Way External Merge Sort

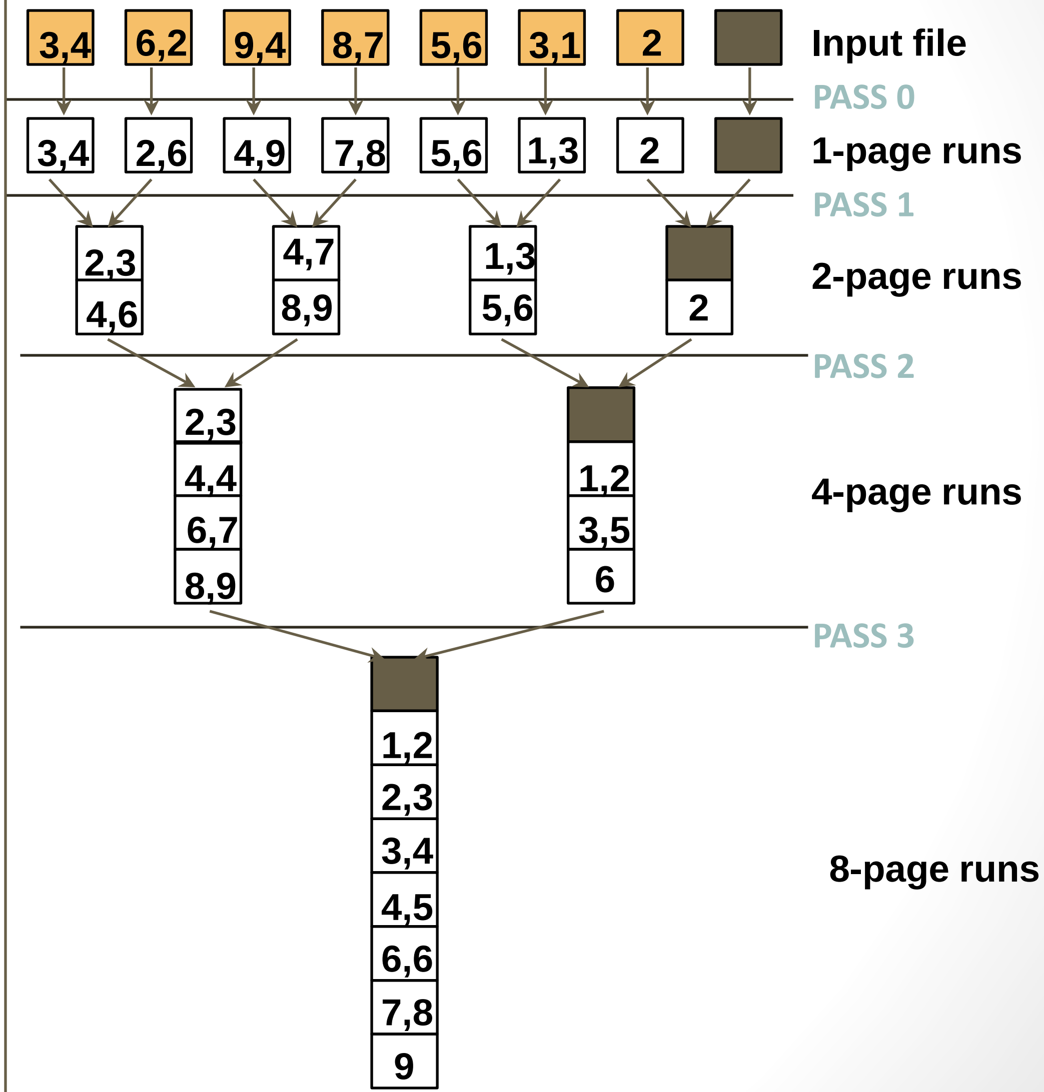
- *Divide and conquer*, sort subfiles (runs) and merge

- Each pass, we read + write N pages in file → 2N.
- Number of passes is:

$$\lceil \log_2 N \rceil + 1$$

- So total cost is:

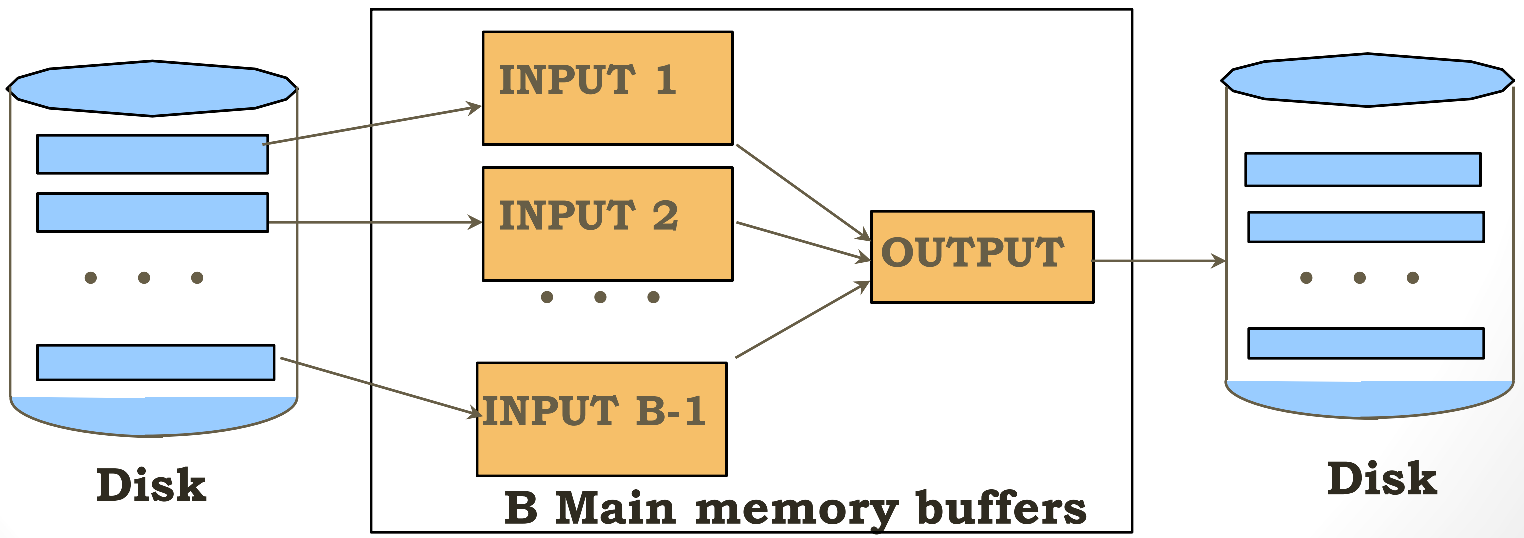
$$2N\left(\lceil \log_2 N \rceil + 1\right)$$



General External Merge Sort

More than 3 buffer pages. How can we utilize them?

- To sort a file with *N* pages using *B* buffer pages:
  - Pass 0: use *B* buffer pages. Produce $\lceil N/B \rceil$ sorted runs of *B* pages each.
  - Pass 2, 3…, etc.: merge *B-1* runs.



Cost of External Merge Sort

E.g., with 5 (B) buffer pages, sort 108 (N) page file:

Pass 0	$\lceil 108/5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)	$\lceil N/B \rceil$ sorted runs of B pages each
Pass 1	$\lceil 22/4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)	$\lceil N/B \rceil / (B-1)$ sorted runs of $B(B-1)$ pages each
Pass 2	2 sorted runs, 80 pages and 28 pages	$\lceil N/B \rceil / (B-1)^2$ sorted runs of $B(B-1)^2$ pages
Pass 3	Sorted file of 108 pages	$\lceil N/B \rceil / (B-1)^3$ sorted runs of $B(B-1)^3$ ($\geq N$) pages

- Number of passes = $1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

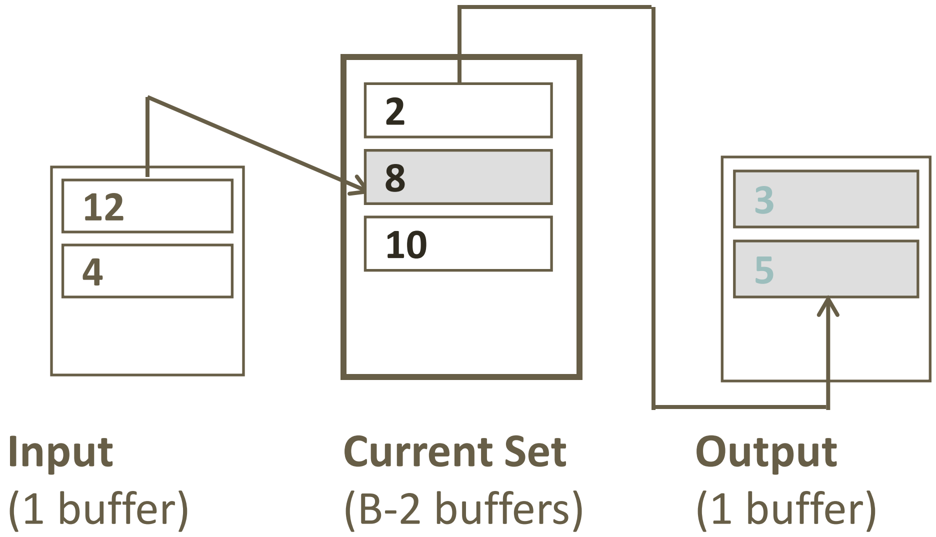
  Cost = 2N * (# of passes)

Number of Passes of External Sort

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

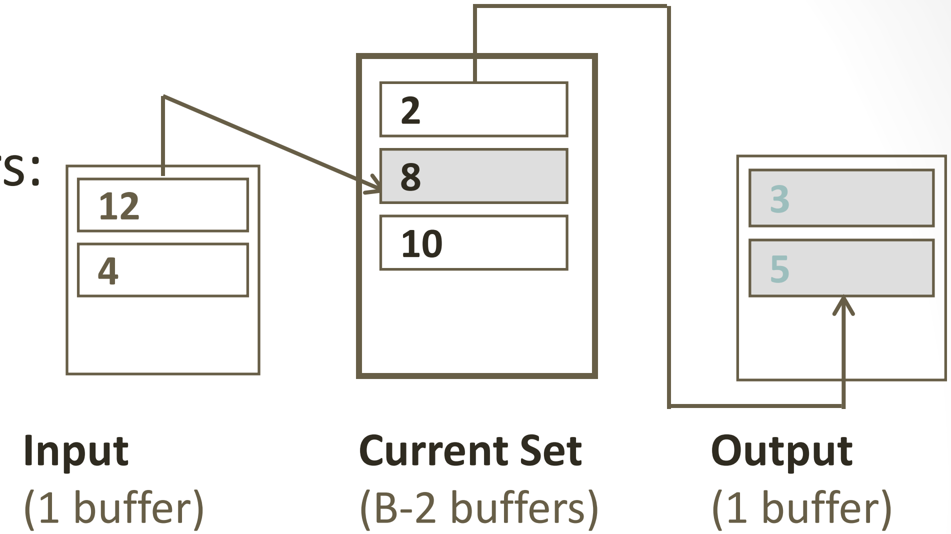
Replacement Sort

- Produces initial sorted runs as long as possible.
- *Replacement Sort*: when used in Pass 0 for sorting, can write out sorted runs of size 2B on average.
  - Affects calculation of the number of passes accordingly.



Replacement Sort

- Organize B available buffers:
  - 1 buffer for *input*
  - B-2 buffers for *current set*
  - 1 buffer for *output*

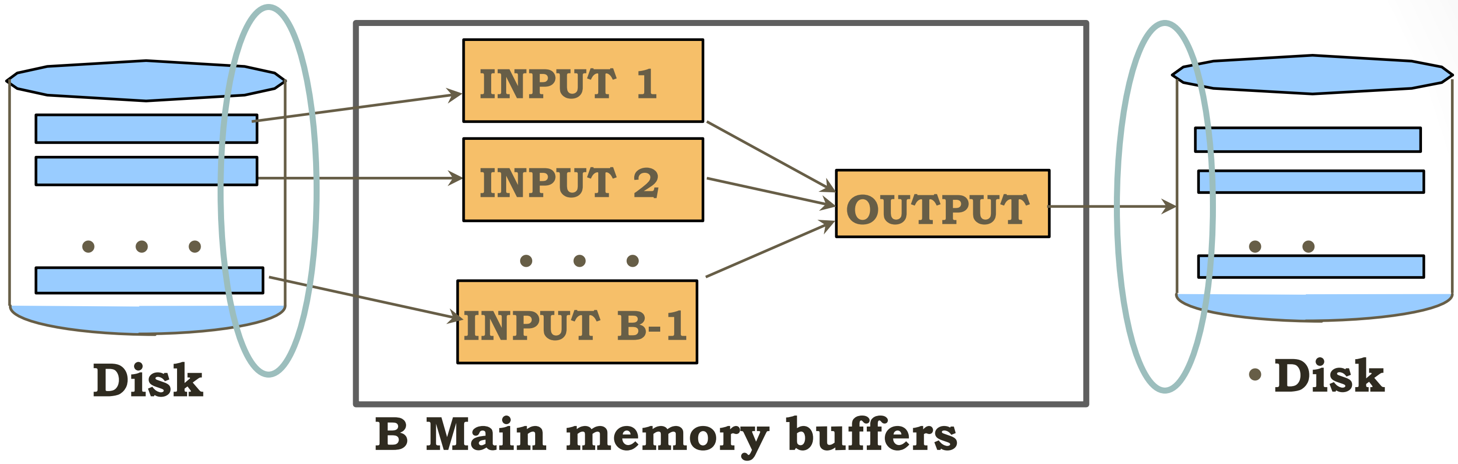


- ❖ Pick tuple *r* in the current set with the *smallest value that is ≥ largest value in output*, e.g. 8, to extend the current run.
- ❖ Fill the space in current set by adding tuples from input.
- ❖ Write output buffer out if full, extending the current run.
- ❖ Current run terminates if *every tuple in the current set is smaller than the largest tuple in output*.

I/O Cost versus Number of I/Os

- Cost metric has so far been the number of I/Os.
- Issue 1: effect of sequential (blocked) I/O?
  - Refine external sorting using *blocked I/O*
- Issue 2: parallelism between CPU and I/O?
  - Refine external sorting using *double buffering*

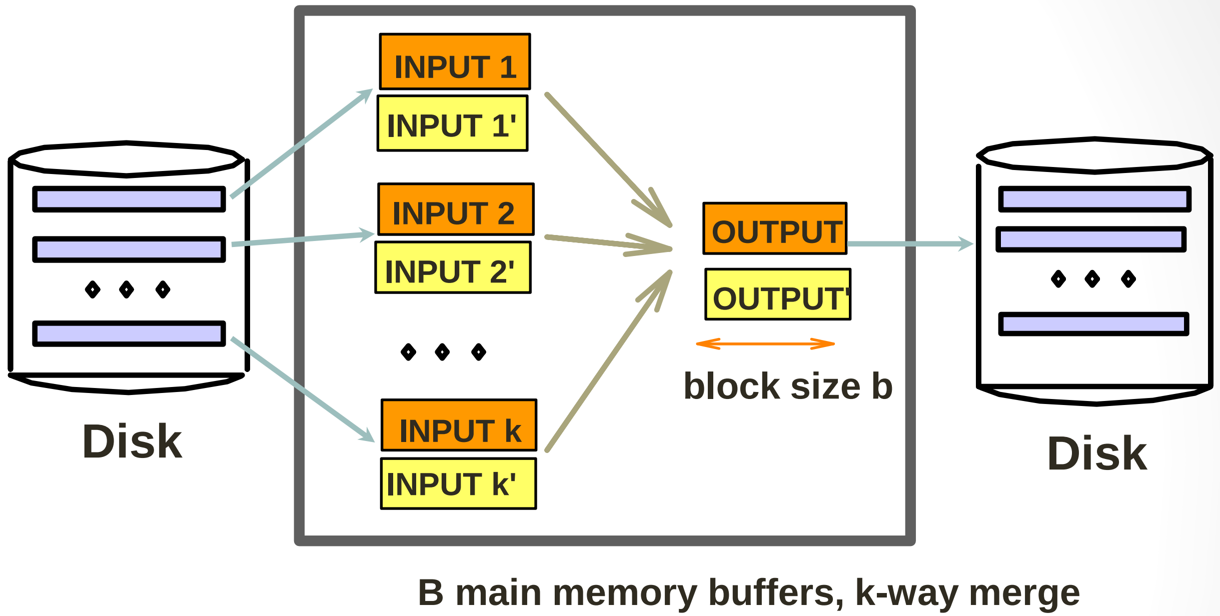
Blocked I/O for External Merge Sort



- *Disk behavior* of external sorting: *sequential* or *random* I/O for input, output?
- To reduce I/O cost, make each input buffer a *block* of pages.
  - But this will reduce fan-out during merge passes! E.g. from B-1 inputs to (B-1)/2 inputs.
  - In practice, most files still sorted in 2-3 passes.

Double Buffering

What happens when an input block has been consumed?



- To reduce wait time for I/O request to complete, can *prefetch* into `shadow block'.
  - Potentially, more passes.
  - In practice, most files *still* sorted in 2-3 passes.

Sorting Records

- Sorting has become a big game
  - Parallel sorting is the name of the game ...
- Datamation sort benchmark: Sort 1M records of size 100 bytes
  - Typical DBMS: 15 minutes
  - World record: 1.18 ***seconds*** (1998 record)
    - 16 off-the-shelf PC, each with 2 Pentium processor, two hard disks, running NT4.0.
    - http://www.berkeley.edu/news/berkeleyan/1999/0120/sort.html
- New benchmarks proposed:
  - Minute Sort: How many can you sort in 1 minute?
  - Dollar Sort: How many can you sort for $1.00?

Using B+ Trees for Sorting

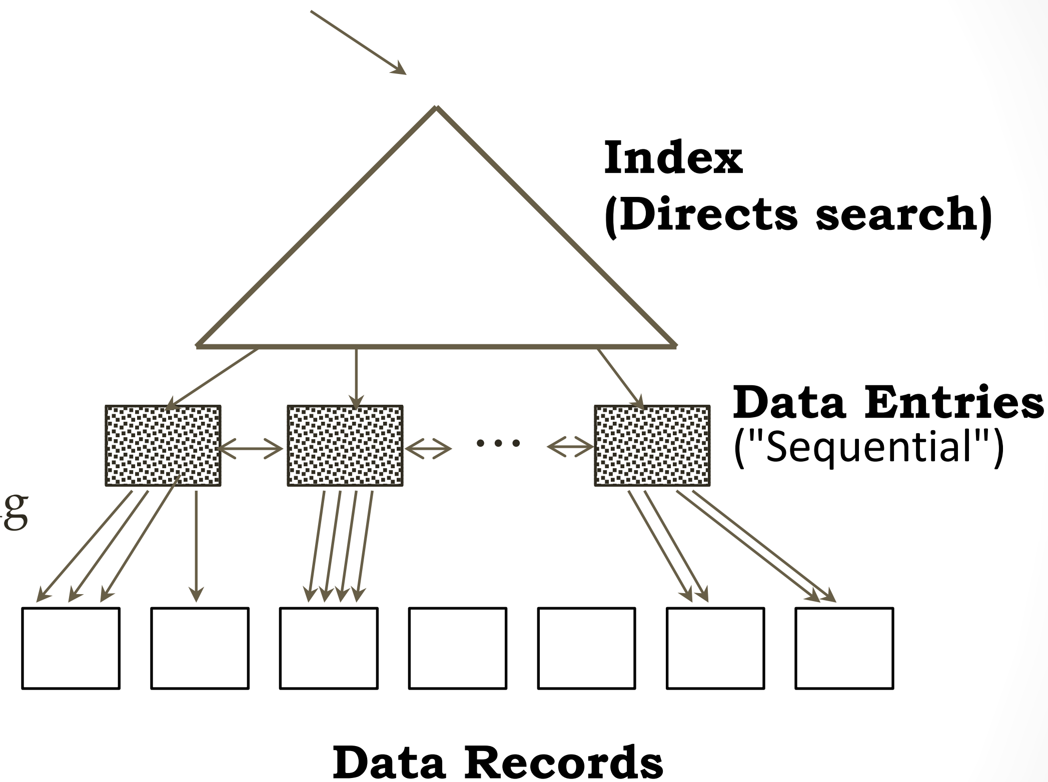
- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- Is this a good idea? Cases to consider:
  - B+ tree is clustered
  - B+ tree is not clustered

Good idea!

Could be a very bad idea!

Clustered B+ Tree Used for Sorting

- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)



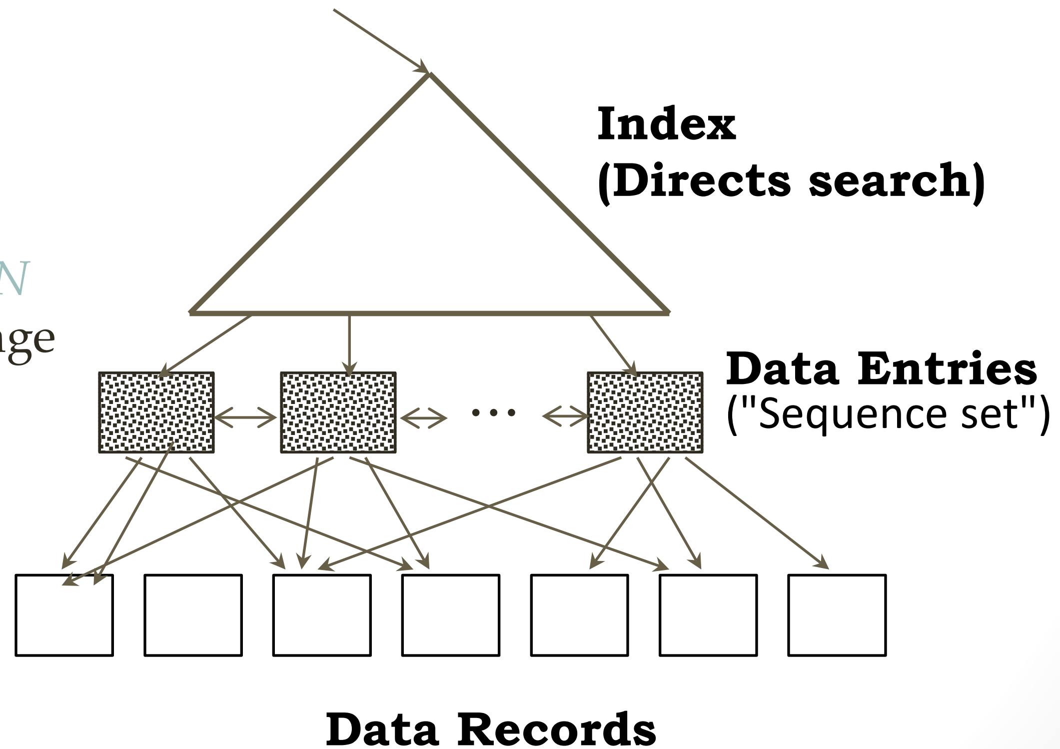
- If Alternative 2 is used? Additional cost of retrieving data records: each page fetched just once.

Almost always better than external sorting!

Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, one I/O per data record!

Worse case I/O: RN
R: # records per page
N: # pages in file



External Sorting vs. Unclustered Index

N	Sorting	R=1	R=10	R=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

For sorting
B=1,000
Block size=32

R: # of records per page
R=100 is the more realistic value.
Worse case numbers (RN) here

Summary: External Sorting

- External sorting is important; DBMS may dedicate part of buffer pool for sorting
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages). Later passes: ***merge*** runs.
  - # of runs merged at a time depends on ***B,*** and ***block size.***
  - Larger block size means less I/O cost per page.
  - Larger block size means smaller # runs merged.
  - In practice, # of passes rarely more than 2 or 3.
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.

Sort-Merge Join Algorithm

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

- Sort R and S on join column using external sorting.
- Merge R and S on join column, output result tuples.

  Repeat until either R or S is finished:
  - *Scanning*:
    - Advance scan of R until current R-tuple >=current S tuple,
    - Advance scan of S until current S-tuple>=current R tuple;
    - Do this until current R tuple = current S tuple.
  - *Matching*:
    - Match all R tuples and S tuples with same value; output <r, s> for all pairs of such tuples.
- Data access patterns for R and S?

  R is scanned once, each S partition scanned once per matching R tuple

Sort-Merge Join

R Sid
28
28
31
31
31
31
58

Q Sid
22
28
31
44
58

Output

28 28

28 28

31 31

31 31

31 31

31 31

58 58

R has multiple matches
Has foreign key to Q

Find a match

Walk right relation for more matches

Walk left Relation for more Matches

```
/* Stage 1: Sorting */
sort R on R.A
sort Q on Q.B

/* Stage 2: Merging */
r = first tuple in R
q = first tuple in Q

while r ≠ EOR and q ≠ EOR do
    if r.A > q.B then
        q = next tuple in Q after q
    else
    if r.A < q.B then
        r = next tuple in R after r
    else
        put r ∘ q in the output relation

        /* output further tuples that match with r */
        q' = next tuple in Q after q
        while q' ≠ EOR and r.A = q'.B do
            put r ∘ q' in the output relation
            q' = next tuple in Q after q'
        od

        /* output further tuples that match with q */
        r' = next tuple in R after r
        while r' ≠ EOR and r'.A = q.B do
            put r' ∘ q in the output relation
            r' = next tuple in R after r'
        od

        r = next tuple in R after r
        q = next tuple in Q after q
    fi
od
```

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

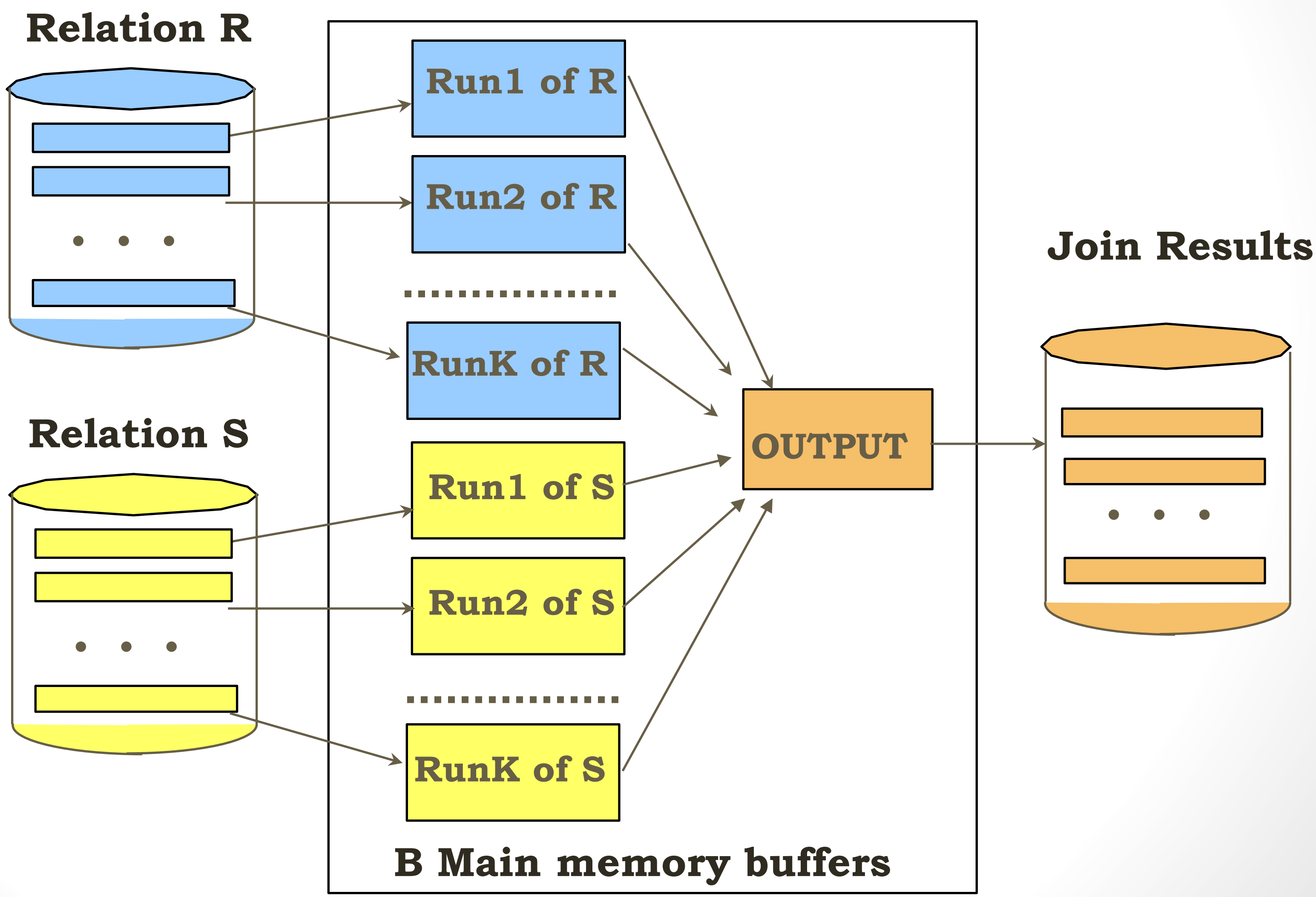
sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

- Cost: M log M + N log N + merging_cost ($\in$[M+N, M*N])
  - The cost of merging could be M*N (but quite unlikely). When?
  - M+N is guaranteed in *foreign key join*; treat the referenced relation as inner
  - As with sorting, log M and log N are small numbers, e.g. 3, 4.
- With 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost is 7500 (assuming M+N).

Refinement of Sort-Merge Join

- *Idea*:
  - *Sorting* of R and S has respective merging phases
  - *Join* of R and S also has a merging phase
  - Combine all these merging phases
- Two-pass algorithm for sort-merge join:
  - Pass 0: sort subfiles of R, S individually
  - Pass 1: merge sorted runs of R, merge sorted runs of S, and merge the resulting R and S files as they are generated by checking the join condition.

2-Pass Sort-Merge Algorithm



Memory Requirement and Cost

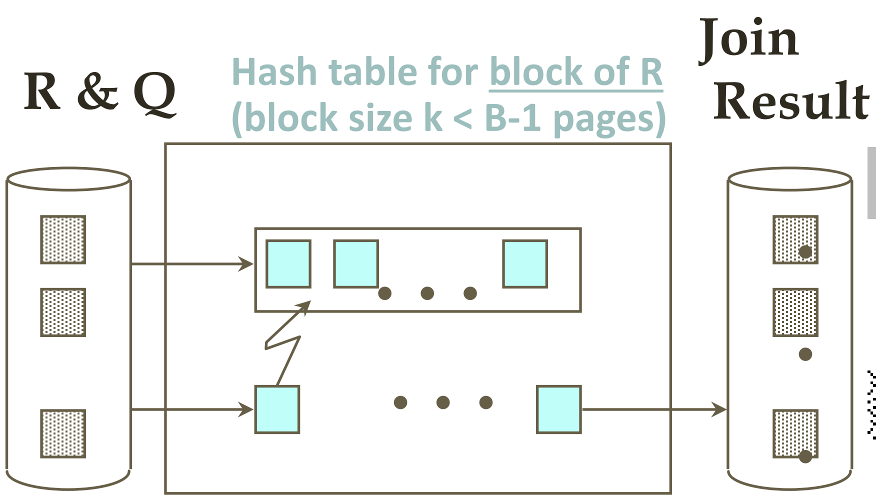
- Memory requirement for 2-pass sort-merge:
  - Assume U is the size of the larger relation. U= max(M, N).
  - Sorting pass produces sorted runs of length up to 2B ("replacement sort") .

    # of runs per relation $\leq$ U/2B.
  - Merging pass holds sorted runs of both relations and an output buffer, merges while checking join condition.

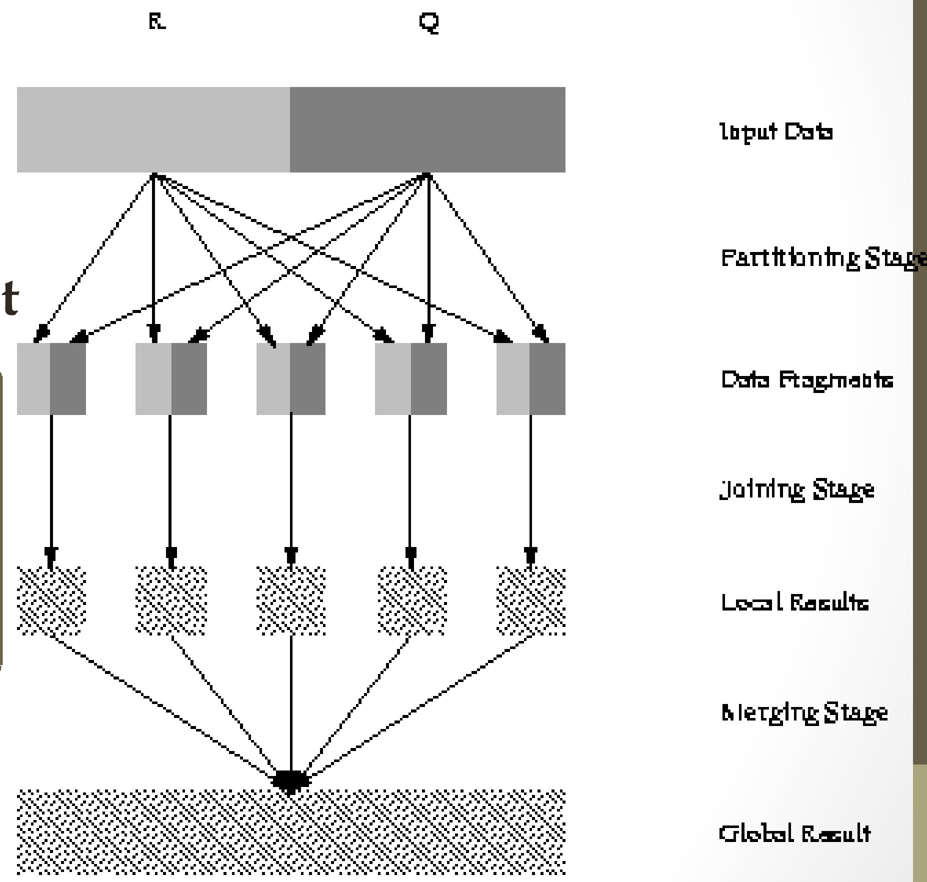
    $2*(U/2B) < B \rightarrow$ $\boxed{\mathbf{B} > \sqrt{U}}$
- Cost: read & write each relation in Pass 0
  + read each relation in merging pass
  (+ writing result tuples, ignore here) = 3 (M+N)
  - In example, cost goes down from 7500 to 4500 I/Os.

Parallelizing Approaches

Fragment and Replica Technique



Symmetric Partititioning



Evaluation of other RAs

- Evaluation of joins
- Evaluation of selections
- Evaluation of projections
- Evaluation of other operations

Using an Index for Selections

- Cost depends on # qualifying tuples, and *clustering*.
  - Cost of finding data entries (often small) + cost of retrieving records (could be large w/o clustering).
  - For *gpa* > 3.0, if 10% of tuples qualify (100 pages, 10,000 tuples), cost ≈ 100 I/Os with a clustered index; otherwise, up to 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.

Each data page is looked at just once, although # of such pages likely to be higher than with clustering.

Approach 1 to General Selections

- (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 is expected to require the smallest # 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 I/O cost.
  - 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.

Approach 2: Intersection of Rids

- 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.
  - *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, if DISTINCT is specified.
- 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.
  - 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.
  - # result tuples smaller than input. Difference depends on # of duplicates.
- Cost: In Pass 0, read input 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.
- 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 a 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.
  - E.g. projection on <sid, age>, search key on <sid, age, rating>.

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.
- Hashing based approach to union:
  - Partition R and S using hash function *h*.
  - For each R-partition, build in-memory hash table (using *h2*). Scan S-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.

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. (Can improve upon this by combining sorting and aggregate computation.)
  - Hashing on group-by attributes also works.
  - 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: Query plan

- 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 ops.