External Sorting and Join Algorithms

Introduction to Databases

CompSci 316 Fall 2020

DUKE
COMPUTER SCIENCE

Announcements (Thu. Oct 8)

- HW-5 + Gradiance-3 (Constraints/Triggers)
  - Due now Monday 10/12 -- extended
- Keep working on your project!
  - MS-2 due next week (10/15)
  - Need to submit a basic working version of your website (all functionalities not needed, but interactions from/to UI and databases should be there) + other things
- Will focus on projects in the discussion session on Monday
- Midterm survey due Tue 10/13

Notation

Recall our disk-memory diagram
On board!

- Relations: R, S
- Tuples: r, s
- Number of tuples: $|R|$, $|S|$
- Number of disk blocks: $B(R)$, $B(S)$
- Number of memory blocks available: M
- Cost metric
  - Number of I/O's
  - Memory requirement

Scanning-based algorithms

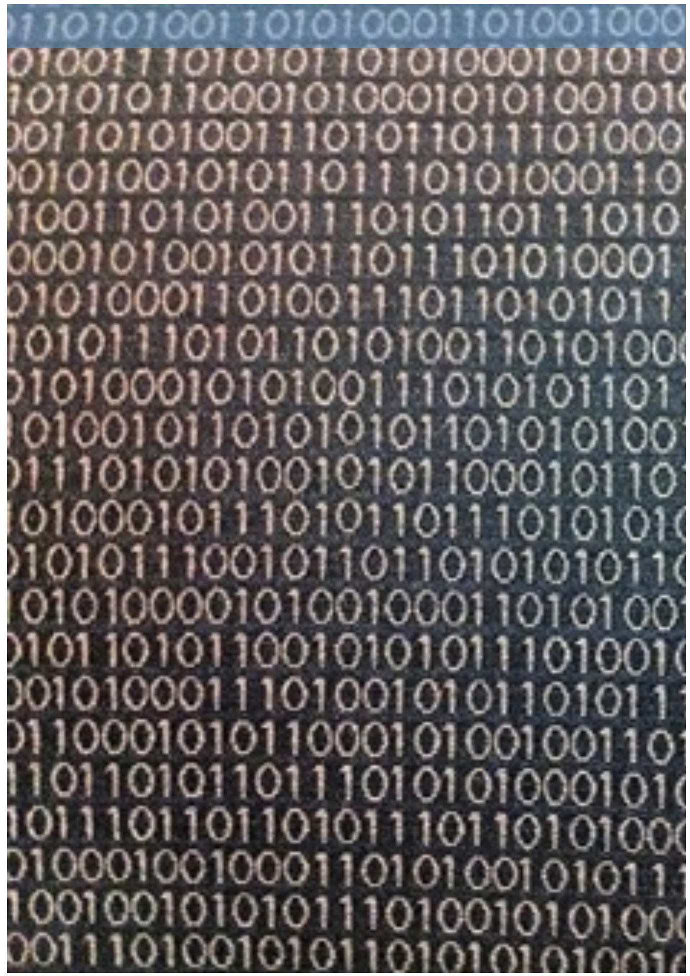


Table scan

- Scan table *R* and process the query
  - Selection over *R*
  - Projection of *R* without duplicate elimination
- I/O's: $B(R)$
  - Trick for selection: stop early if it is a lookup by key
- Memory requirement: 2
- Not counting the cost of writing the result out
  - Same for any algorithm!
  - Maybe not needed—results may be pipelined into another operator

Announcements (Tue. Oct 13)

- Keep working on your project!
  - MS-2 due next Monday (10/19)
  - Need to submit a basic working version of your website (all functionalities not needed, but interactions from/to UI and databases should be there) + other things

- Midterm survey due today Tue 10/13

- HW6 to be posted today, due next Thursday

- How do we implement Join?

- Cost?
  - (page I/O -- in terms of B(R), |R| etc.)

- Memory requirement?

Nested-loop join

$R \bowtie_p S$

- For each block of R, and for each r in the block:
  For each block of S, and for each s in the block:
  Output rs if p evaluates to true over r and s
  - R is called the outer table; S is called the inner table
  - I/O's: $B(R) + |R| \cdot B(S)$
  - Memory requirement: 3

Improvement: block-based nested-loop join

Block-based Nested Loop Join

- $R \bowtie_p S$
- R outer, S inner
- For each block of R, for each block of S:
  For each r in the R block, for each s in the S block: …
  - I/O's: $B(R) + B(R) \cdot B(S)$
  - Memory requirement: same as before

More improvements

- Make use of available memory
  - Stuff memory with as much of R as possible, stream S by, and join every S tuple with all R tuples in memory
  - I/O's: $B(R) + \left\lceil \frac{B(R)}{M-2} \right\rceil \cdot B(S)$
    - Or, roughly: $B(R) \cdot B(S)/M$
  - Memory requirement: M (as much as possible)

- Which table would you pick as the outer?

Sorting-based algorithms

http://en.wikipedia.org/wiki/Mail_sorter#mediaviewer/File:Mail_sorting,1951.jpg

See example on the next slide first

External merge sort

Remember (internal-memory) merge sort?
Problem: sort R, but R does not fit in memory

To Understand:

What is a run?
What is a level and a pass?

Reminder: How 2-way merge sort works?
How to extend to multi-way merge sort?

External merge sort

Remember (internal-memory) merge sort?

Problem: sort R, but R does not fit in memory

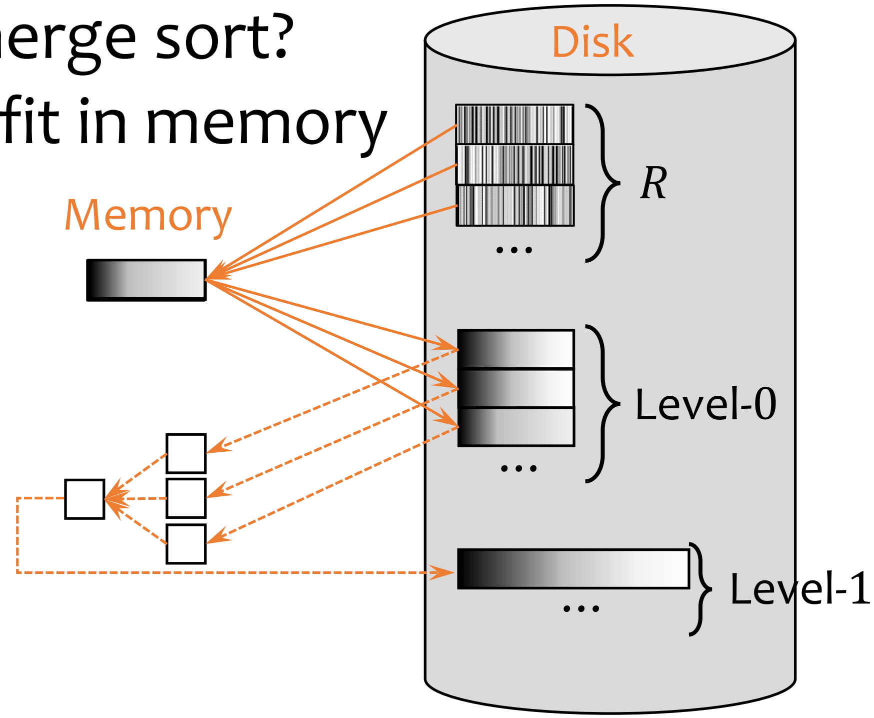
- Pass 0: read M blocks of R at a time, sort them, and write out a level-0 run

- Pass 1: merge $(M - 1)$ level-0 runs at a time, and write out a level-1 run

- Pass 2: merge $(M - 1)$ level-1 runs at a time, and write out a level-2 run

…

- Final pass produces one sorted run



Toy example

- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 3, 6, 9
- Pass 0
  - 1, 7, 4 → 1, 4, 7
  - 5, 2, 8 → 2, 5, 8
  - 9, 6, 3 → 3, 6, 9
- Pass 1
  - 1, 4, 7 + 2, 5, 8 → 1, 2, 4, 5, 7, 8
  - 3, 6, 9
- Pass 2 (final)
  - 1, 2, 4, 5, 7, 8 + 3, 6, 9 → 1, 2, 3, 4, 5, 6, 7, 8, 9

Analysis

- Pass 0: read M blocks of R at a time, sort them, and write out a level-0 run
  - There are $\left\lceil \frac{B(R)}{M} \right\rceil$ level-0 sorted runs
- Pass i: merge $(M - 1)$ level-$(i - 1)$ runs at a time, and write out a level-i run
  - $(M - 1)$ memory blocks for input, 1 to buffer output
  - # of level-i runs = $\left\lceil \frac{\text{\# of level}-(i-1)\text{ runs}}{M-1} \right\rceil$
- Final pass produces one sorted run

Note: The pages of memory are being reused!

- We just have M memory blocks/pages, whereas the number of blocks of R can be much larger
  - B(R) >> M typically
  - Otherwise you will load all pages and sort in memory in a single pass!

- We need to reuse both input and output pages in memory
  - Once the output pages are full, flush them (write) to disk
  - Once an input page is fully processed in Pass-1 onward, get the next page from the same run
  - In pass-0, sort M-pages together, reuse the memory pages for the next M-pages and so on…

- Pass-0 uses an "in-place" sorting algorithm (with constant additional space), so all M pages can be used

Performance of external merge sort

- Number of passes: $\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil + 1$
- I/O's
  - Multiply by $2 \cdot B(R)$: each pass reads the entire relation once and writes it once
  - Subtract $B(R)$ for the final pass
  - Roughly, this is $O\big(B(R) \times \log_M B(R)\big)$
- Memory requirement: M (as much as possible)

We do not count the final write!

Some tricks for sorting

- Double buffering
  - Allocate an additional block for each run
  - Overlap I/O with processing
  - Trade-off: smaller fan-in (more passes)
- Blocked I/O
  - Instead of reading/writing one disk block at time, read/write a bunch ("cluster")
  - More sequential I/O's
  - Trade-off: larger cluster → smaller fan-in (more passes)

Announcements (Thu. Oct 15)

- Keep working on your project!
  - MS-2 due next Monday (10/19)
  - Need to submit a basic working version of your website (all functionalities not needed, but interactions from/to UI and databases should be there) + other things
- HW6 due next Thursday (10/22)
- Short Lecture-quiz-3 (Sorting etc.) due next Thursday (10/22)
- No Gradiance this week.
- Review of clustered/unclustered on Monday

Sort-merge join

$R \bowtie_{R.A=S.B} S$

- Sort R and S by their join attributes; then merge

  r, s = the first tuples in sorted R and S

  Repeat until one of R and S is exhausted:

  If $r.A > s.B$ then s = next tuple in S

  else if $r.A < s.B$ then r = next tuple in R

  else output all matching tuples, and

  r, s = next in R and S

- I/O's: sorting $+ 2B(R) + 2B(S)$ (always?)
  - In most cases (e.g., join of key and foreign key)
  - Worst case is $B(R) \cdot B(S)$: everything joins

Example of merge join

R:

$r_1.A = 1$

$r_2.A = 3$

$r_3.A = 3$

$r_4.A = 5$

$r_5.A = 7$

$r_6.A = 7$

$r_7.A = 8$

S:

$s_1.B = 1$

$s_2.B = 2$

$s_3.B = 3$

$s_4.B = 3$

$s_5.B = 8$

$R \bowtie_{R.A=S.B} S$:

$r_1 s_1$

$r_2 s_3$

$r_2 s_4$

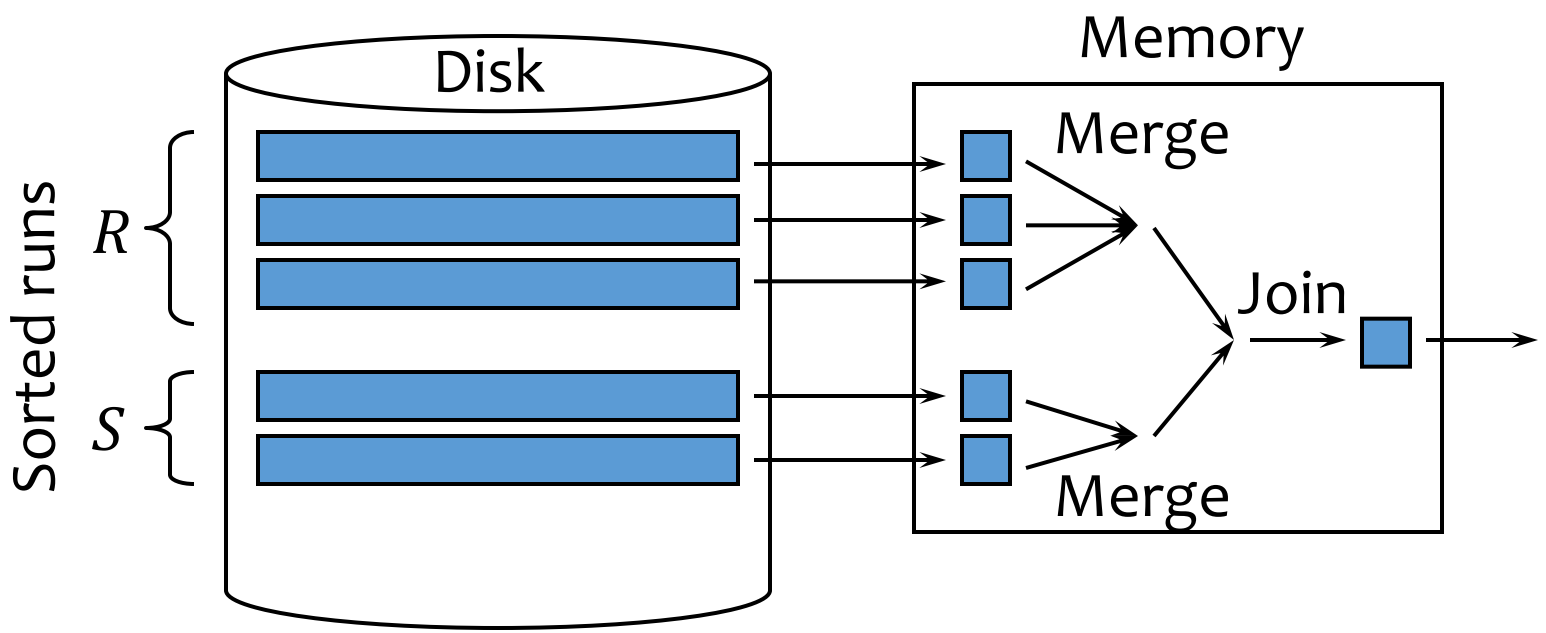
$r_3 s_3$

$r_3 s_4$

$r_7 s_5$

Optimization of SMJ

- Idea: combine join with the (last) merge phase of merge sort
- Sort: produce sorted runs for R and S such that there are fewer than M of them total
- Merge and join: merge the runs of R, merge the runs of S, and merge-join the result streams as they are generated!



Performance of SMJ

- If SMJ completes in two passes:
  - I/O's: $3 \cdot \left(B(R) + B(S)\right)$ - why 3?
  - Memory requirement
    - We must have enough memory to accommodate one block from each run: $M > \frac{B(R)}{M} + \frac{B(S)}{M}$
    - $M > \sqrt{B(R) + B(S)}$
- If SMJ cannot complete in two passes:
  - Repeatedly merge to reduce the number of runs as necessary before final merge and join

Other sort-based algorithms

- Union (set), difference, intersection
  - More or less like SMJ
- Duplication elimination
  - External merge sort
    - Eliminate duplicates in sort and merge
- Grouping and aggregation
  - External merge sort, by group-by columns
    - Trick: produce "partial" aggregate values in each run, and combine them during merge
      - This trick doesn't always work though
        - Examples: SUM(DISTINCT …), MEDIAN(…)

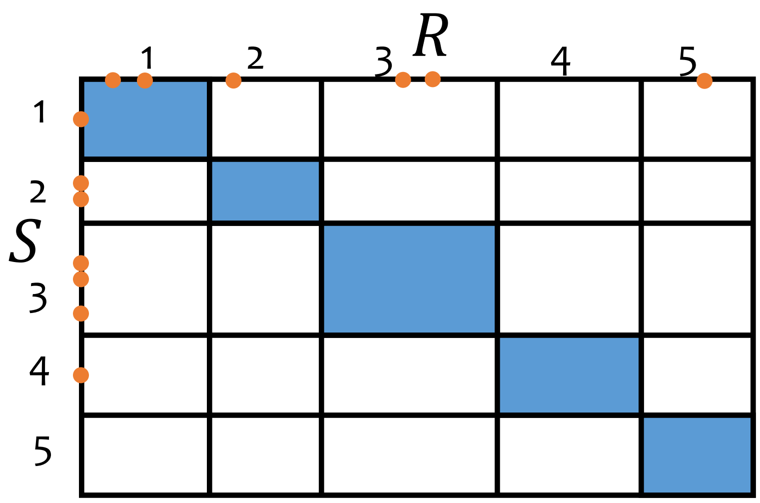
Hashing-based algorithms

http://global.rakuten.com/en/store/citygas/item/041233/

Hash join

$R \bowtie_{R.A=S.B} S$

- Main idea
  - Partition R and S by hashing their join attributes, and then consider corresponding partitions of R and S
  - If $r.A$ and $s.B$ get hashed to different partitions, they don't join

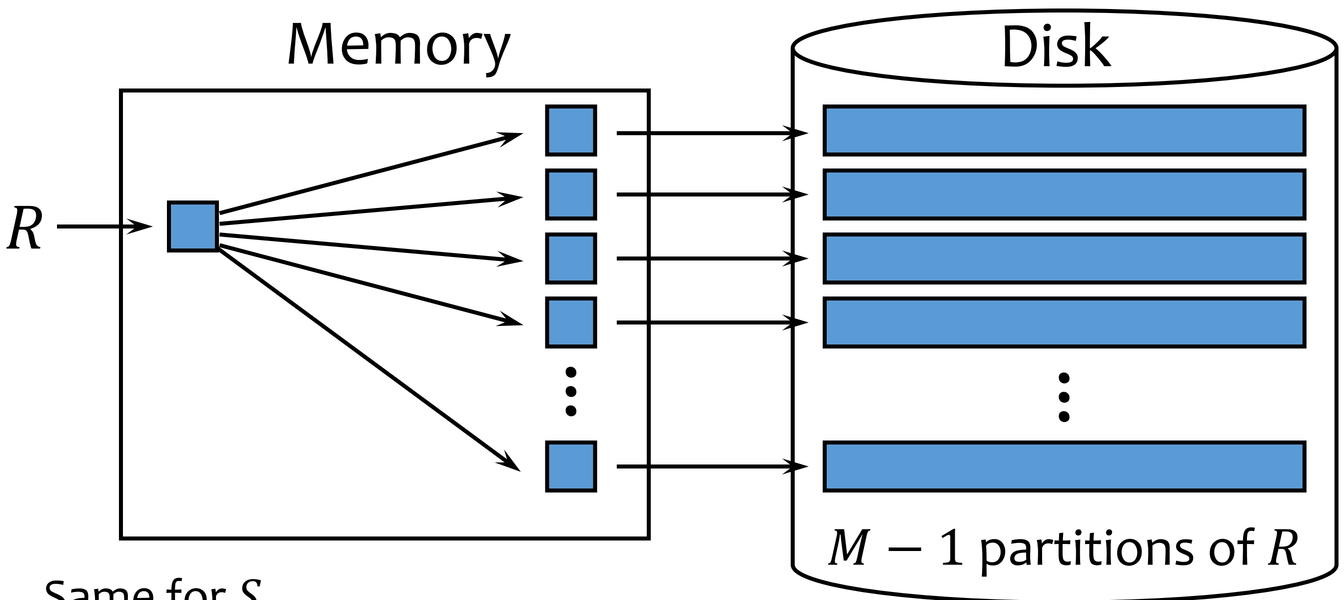


Nested-loop join considers all slots

Hash join considers only those along the diagonal!

Partitioning phase

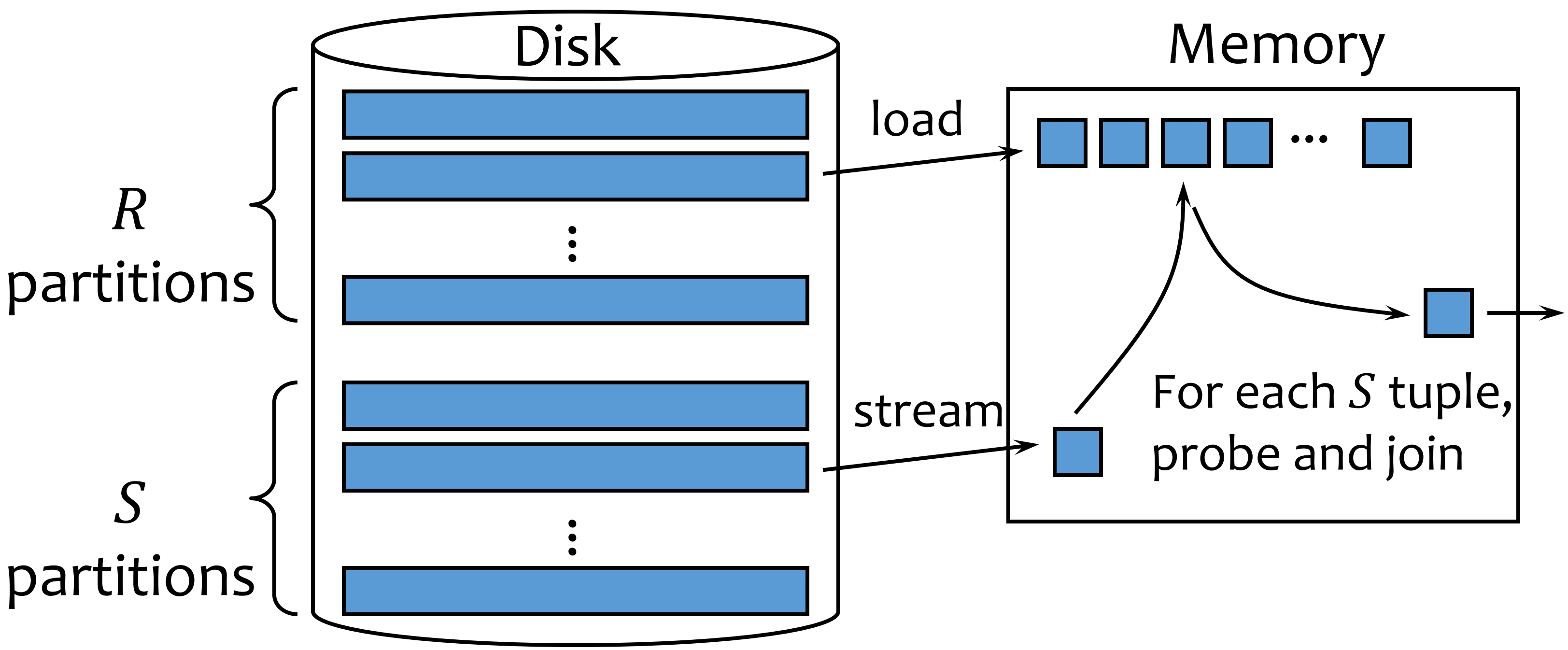
- Partition R and S according to the same hash function on their join attributes



Same for S

Probing phase

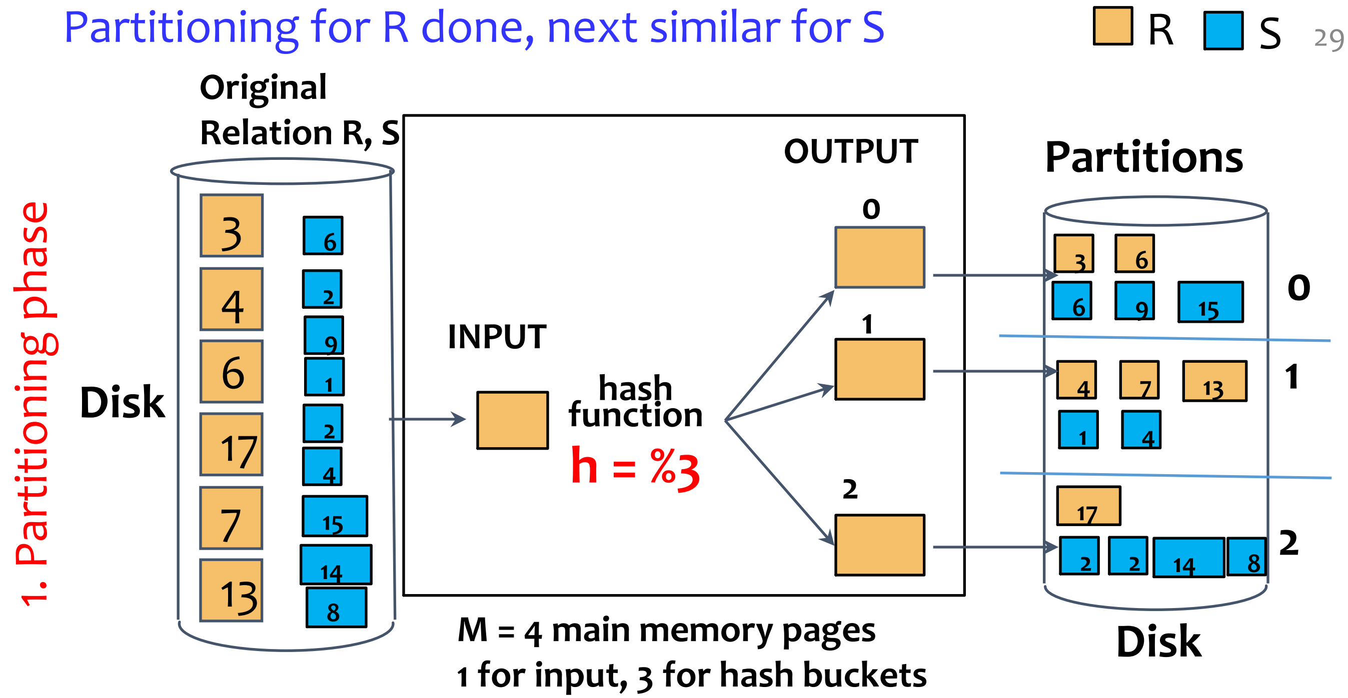
- Read in each partition of R, stream in the corresponding partition of S, join
  - Typically build a hash table for the partition of R
    - Not the same hash function used for partition, of course!



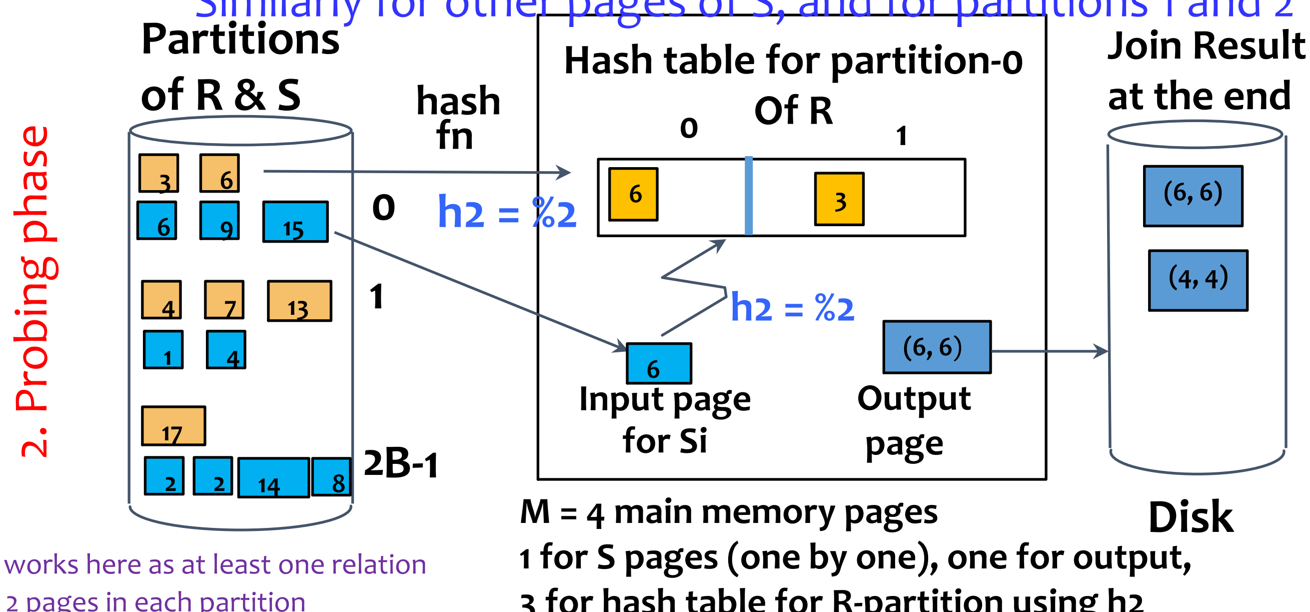
Partitioning for R done, next similar for S

Example

- R(A), S(B)
- $R \bowtie_{R.A=S.B} S$
- B(R) = 6
- B(S) = 9
- M = 4
- Each page of R, S contains just one record
- Hash function for partitioning h = A % 3 (for R), B% 3 for S
- Hash function for probing h2 = A % 2 (for R), B% 2 for S



Probing for partition-0 and 1st page of S in partition 0, Similarly for other pages of S, and for partitions 1 and 2



2-pass works here as at least one relation has <= 2 pages in each partition

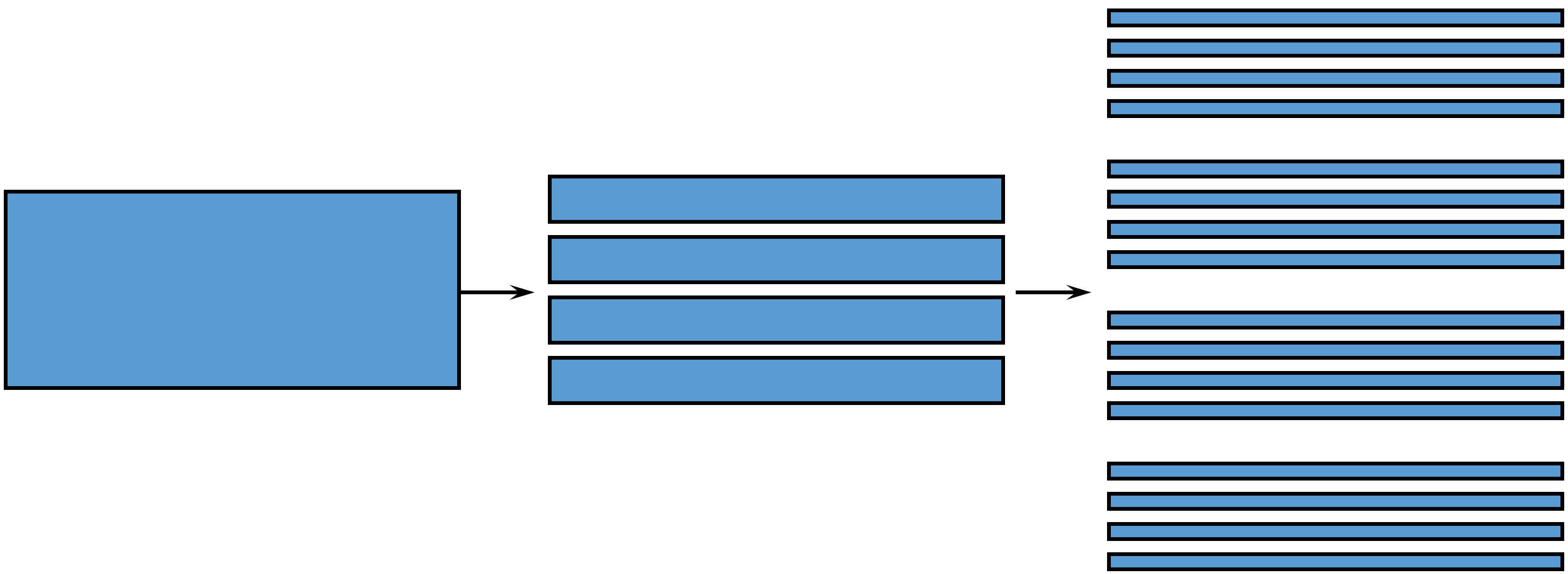
Performance of (two-pass) hash join

- If hash join completes in two passes:
  - I/O's: $3 \cdot \left(B(R) + B(S)\right)$
  - Memory requirement:
    - In the probing phase, we should have enough memory to fit one partition of R: $M - 1 > \frac{B(R)}{M-1}$
    - $M > \sqrt{B(R)} + 1$
    - We can always pick R to be the smaller relation, so:

$$M > \sqrt{\min\left(B(R), B(S)\right)} + 1$$

Generalizing for larger inputs

- What if a partition is too large for memory?
  - Read it back in and partition it again!
    - See the duality in multi-pass merge sort here?



Hash join versus SMJ

(Assuming two-pass)

- I/O's: same
- Memory requirement: hash join is lower
  - $\sqrt{\min\big(B(R), B(S)\big)} + 1 < \sqrt{B(R) + B(S)}$
  - Hash join wins when two relations have very different sizes
- Other factors
  - Hash join performance depends on the quality of the hash
    - Might not get evenly sized buckets
  - SMJ can be adapted for inequality join predicates
  - SMJ wins if R and/or S are already sorted
  - SMJ wins if the result needs to be in sorted order

What about nested-loop join?

- May be best if many tuples join
  - Example: non-equality joins that are not very selective
- Necessary for black-box predicates
  - Example: WHERE *user_defined_pred*$(R.A, S.B)$

Announcements (Tue. Oct 20)

- HW6a (probs 1, 2) due Thursday (10/22)
- HW6b (prob 3) due next Tuesday (10/27)

- Short Lecture-quiz-3 (Sorting etc.) due next Thursday (10/22)

- No Gradiance this week.

- Review of keys/superkeys/FDs/BCNF on Monday

- Please check all grades posted – regrade requests through gradescope or Google Form only within a week

Other hash-based algorithms

- Just like Sorting!

- Union (set), difference, intersection
  - More or less like hash join
- Duplicate elimination
  - Check for duplicates within each partition/bucket
- Grouping and aggregation
  - Apply the hash functions to the group-by columns
  - Tuples in the same group must end up in the same partition/bucket
  - Keep a running aggregate value for each group
    - May not always work

Index-based algorithms

http://i1.trekearth.com/photos/28820/p2270994.jpg

Selection using index

- Equality predicate: $\sigma_{A=v}(R)$
  - Use an ISAM, B+-tree, or hash index on $R(A)$
- Range predicate: $\sigma_{A>v}(R)$
  - Use an ordered index (e.g., ISAM or B+-tree) on $R(A)$
  - Hash index is not applicable

- Indexes other than those on $R(A)$ may be useful
  - Example: B+-tree index on $R(A, B)$
  - How about B+-tree index on $R(B, A)$?

Index versus table scan

Situations where index clearly wins:

- Index-only queries which do not require retrieving actual tuples
  - Example: $\pi_A\left(\sigma_{A>v}(R)\right)$
- Primary index clustered according to search key
  - One lookup leads to all result tuples in their entirety

Index versus table scan (cont'd)

BUT(!):

- Consider $\sigma_{A>v}(R)$ and a secondary, non-clustered index on $R(A)$
  - Need to follow pointers to get the actual result tuples
  - Say that 20% of R satisfies $A > v$
    - Could happen even for equality predicates
  - I/O's for index-based selection: lookup + 20% $|R|$
  - I/O's for scan-based selection: $B(R)$
  - Table scan wins if a block contains more than 5 tuples!

Index nested-loop join

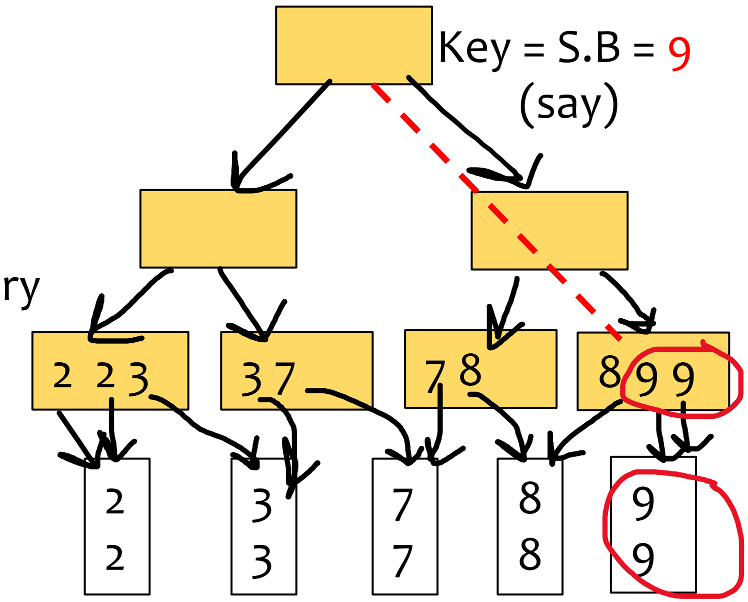
$R \bowtie_{R.A=S.B} S$

- Idea: use a value of $R.A$ to probe the index on $S(B)$
- For each block of R, and for each r in the block:
  - Use the index on $S(B)$ to retrieve s with $s.B = r.A$
    - Output rs
- I/O's: $B(R) + |R| \cdot (\text{index lookup})$
  - Typically, the cost of an index lookup is 2-4 I/O's
  - Beats other join methods if $|R|$ is not too big
  - Better pick R to be the smaller relation
- Memory requirement: 3

Example

Query: $R \bowtie_{R.A=S.B} S$

- R.A values (1 R-tuple/page): 7, 2, 9, 8, 3
  - B(R) = |R| = 5
- B+-tree Index on S.B, 2 S-tuples/data page
  - Clustered, 3 levels, all index/data pages in memory
  - Assume foreign key S.B to primary key R.A
  - Assume each R.A joins with the same no. of S.B
  - |S| = 10, B(S) = 5
  - Assume matching data entries fit in one leaf
  - Each R tuple joins with 2 S tuples that fit in 1 S-page



- Algo:
  - For every page of R

    Cost of R = B(R) = 5
    - For every tuple of R in that page
      - Send the value of R.A as the key value
      - Retrieve the matching S records from data pages pointed to by the matching index entries
      - Output all of them

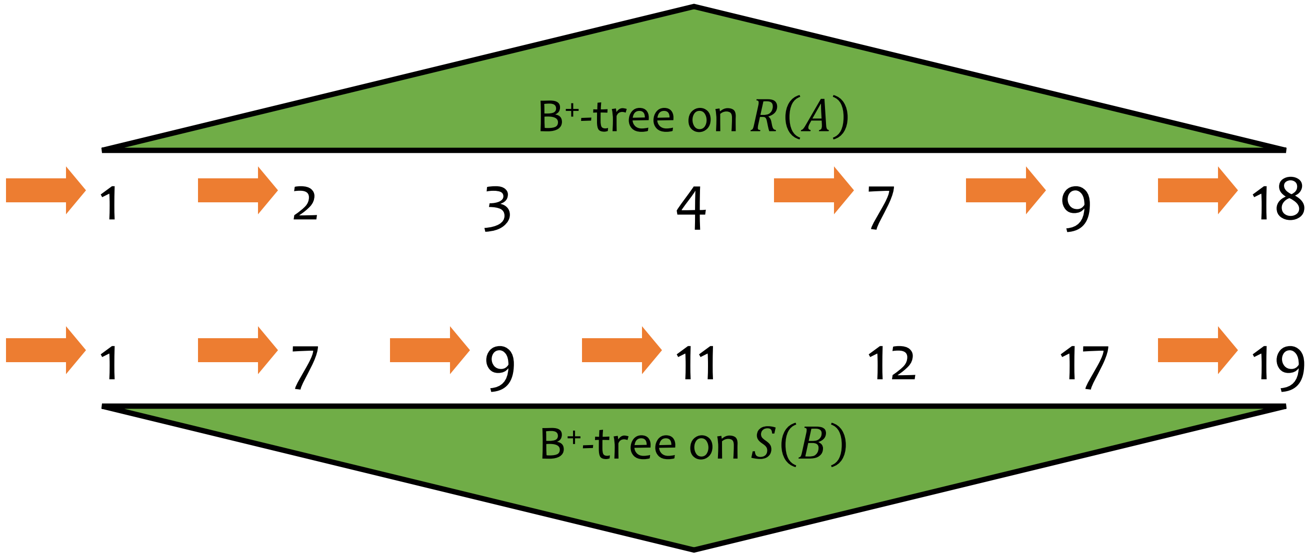
Total cost for S = |R| * (3 + 1)

- For every R.A value, max cost of accessing matching S tuples = 3 (accessing leaves) + 1 (accessing data page)
- Total cost of index-nested-loop-join = B(R) + |R| (3+1) = 5 + 5 * 4 = 25

Zig-zag join using ordered indexes

$R \bowtie_{R.A=S.B} S$

- Idea: use the ordering provided by the indexes on $R(A)$ and $S(B)$ to eliminate the sorting step of sort-merge join
- Use the larger key to probe the other index
    - Possibly skipping many keys that don't match



Summary of techniques

- Scan
  - Selection, duplicate-preserving projection, nested-loop join
- Sort
  - External merge sort, sort-merge join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Hash
  - Hash join, union (set), difference, intersection, duplicate elimination, grouping and aggregation
- Index
  - Selection, index nested-loop join, zig-zag join