SFU

Database Systems I

CMPT 354 Summer 2024

Zhengjie Miao

July 17 2024

Announcements (Wed. July 17)

- A5 released, due July 29

Overview

- Many ways to process the same query
  - Scan? Sort? Hash? Use an index?
  - All have different performance characteristics and/or make different assumptions about data
- Best choice depends on the situation
  - Implement all alternatives
  - Let the query optimizer choose at run-time

Notation

- 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

Overview

- **Scanning-based algorithms**
- Sorting-based algorithms
- Hashing-based algorithms
- Index-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

One is loading the blocks to disk, and the other is writing the results of the query to the disk

Memory should not take into consideration the amount of memory needed to write back the results to disk nor any intermediate values.

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: $B(R) + |R| \cdot B(S)$

Blocks of R are moved into memory only once

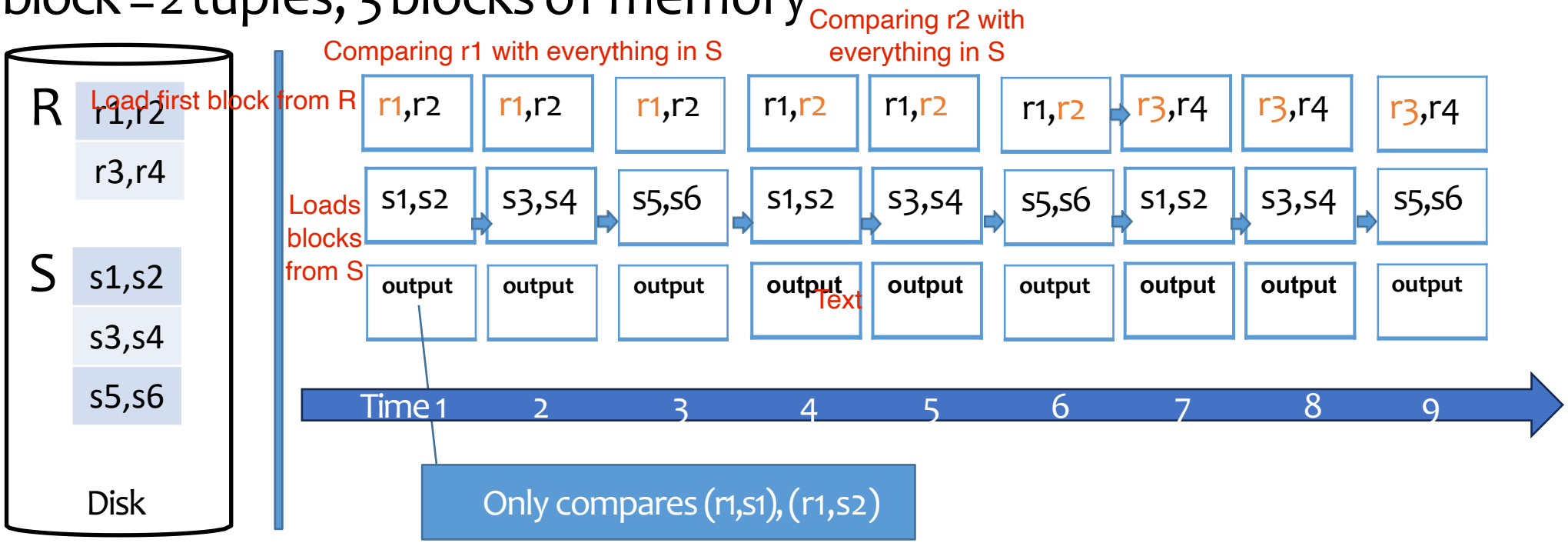
Blocks of S are moved into memory |R| number of times

- Memory requirement: 3

We need one memory block for R, one block to store S, and one block to store the results of the predicate.

Example for basic nested loop join

1 block = 2 tuples, 3 blocks of memory



- Number of I/O:

We can be more efficient

$$B(R) + |R| * S(R) = 2\,\text{blocks} + 4 * 3\,\text{blocks} = 14$$

Improvement: block-based nested-loop join

$R \bowtie_p S$

- For each block B_R of R
  For each block B_S of S:
    For each r in the B_R block
      For each s in the B_S block
        Output rs if p evaluates to true over r and s

- I/O's: $B(R) + B(R) \cdot B(S)$

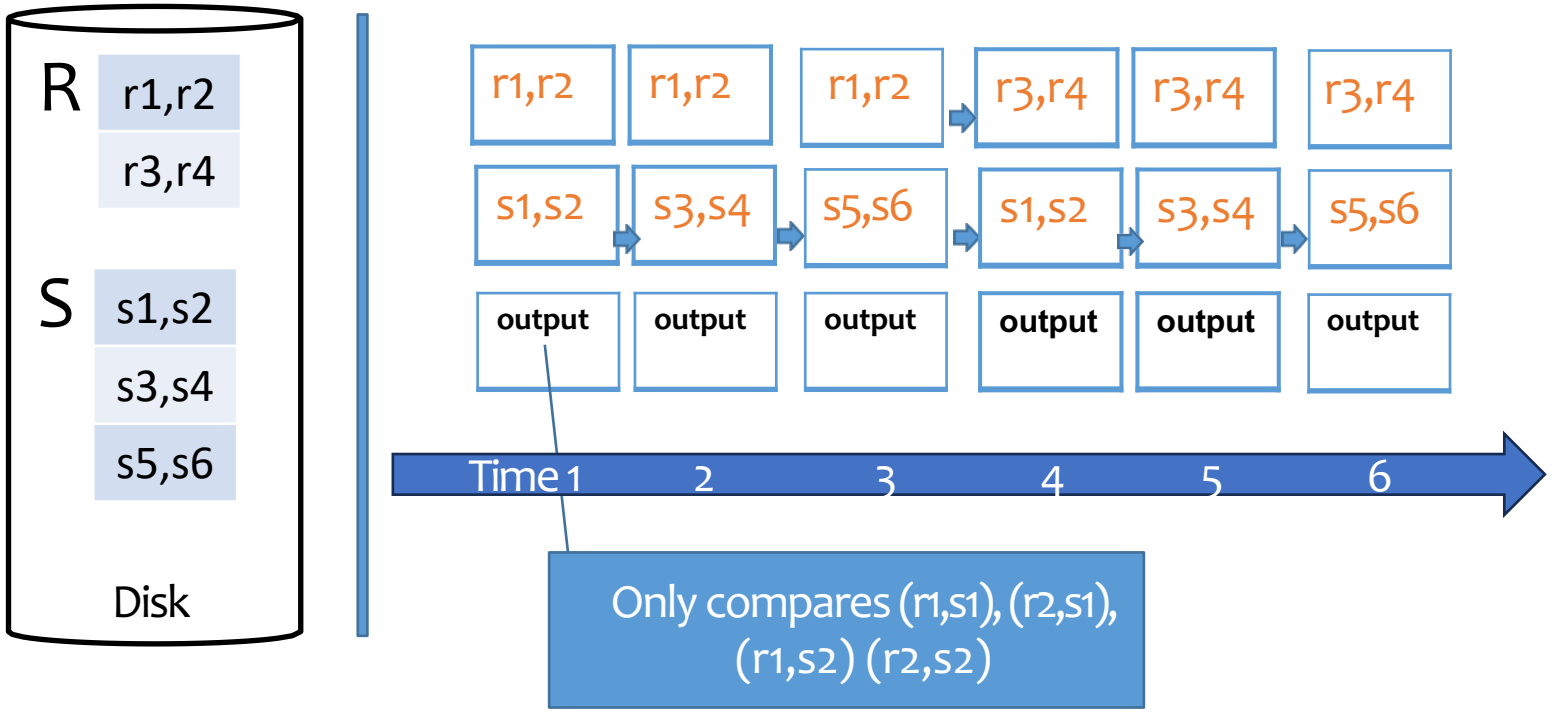
Blocks of R are moved into memory only once

Blocks of S are moved into memory B(R) number of times

- Memory requirement: 3
  - same as before

Example for block-based nested loop join

1 block = 2 tuples, 3 blocks of memory



- Number of I/O:

$$B(R) + |R| * S(R) = 2\,\text{blocks} + 2 * 3\,\text{blocks} = 8$$

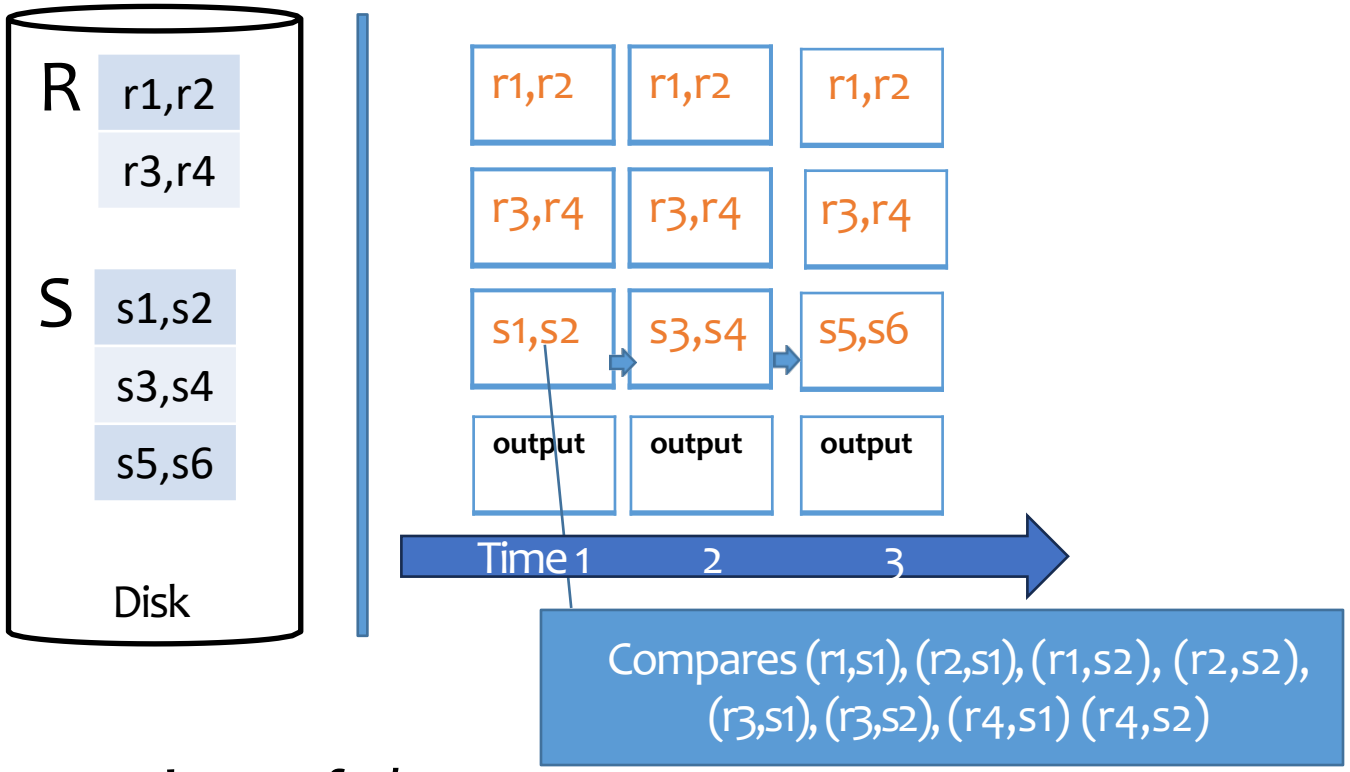
More improvements

- Stop early if the key of the inner table is being matched
- 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?

Put the table that can fit into memory as the inner one?
Ask professor about this.

Example for block-based nested loop join

1 block = 2 tuples, 4 blocks of memory



- Number of I/O:

$$B(R) + B(R)/(M-2) * S(R) = 2\,\text{blocks} + 1 * 3\,\text{blocks} = 5$$

Where does one come from?

Overview

- Scanning-based algorithms
- **Sorting-based algorithms**
- Hashing-based algorithms
- Index-based algorithms

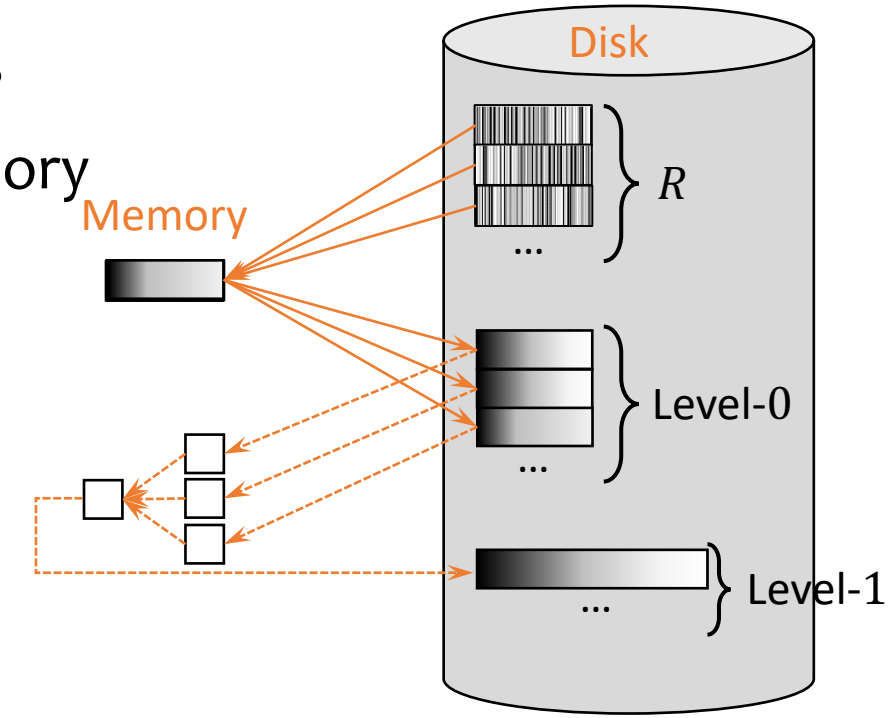
External merge sort

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

…

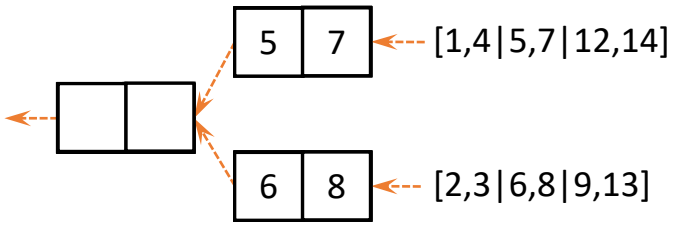
- Final pass produces one sorted run

Toy example

3 memory blocks available; each holds two numbers

Input: [1, 14 | 7, 12 | 4, 5 | 2, 13 | 8, 9 | 6, 3 | 11, 10]

- Pass 0 Since you have three memory blocks, you cannot merge all three blocks at the same time. You need to leave one to output the results
  - [1, 14 | 7, 12 | 4, 5] → [1, 4 | 5, 7 | 12, 14]
  - [2, 13 | 8, 9 | 6, 3] → [2, 3 | 6, 8 | 9, 13]
  - [11, 10] → [10, 11]
- Pass 1
  - [1, 4 | 5, 7 | 12, 14] + [2, 3 | 6, 8 | 9, 13] → [1, 2 | 3, 4 | 5, 6 | 7, 8 | 9, 12 | 13, 14]
  - Leave [10, 11] alone
- Pass 2 (final)
  - [1, 2 | 3, 4 | 5, 6 | 7, 8 | 9, 12 | 13, 14] + [10, 11] → [1, 2 | 3, 4 | 5, 6 | 7, 8 | 9, 10 | 11, 12 | 13, 14]



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

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 $O\left(B(R) \times \log_M B(R)\right)$
- Memory requirement: M (as much as possible)

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

Potentially a mini nested loop

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:

- $s_1.B = 1$
- $s_2.B = 2$
- $s_3.B = 3$
- $s_4.B = 3$
- $s_5.B = 8$

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

$r_1 s_1$

$r_2 s_3$

$r_2 s_4$

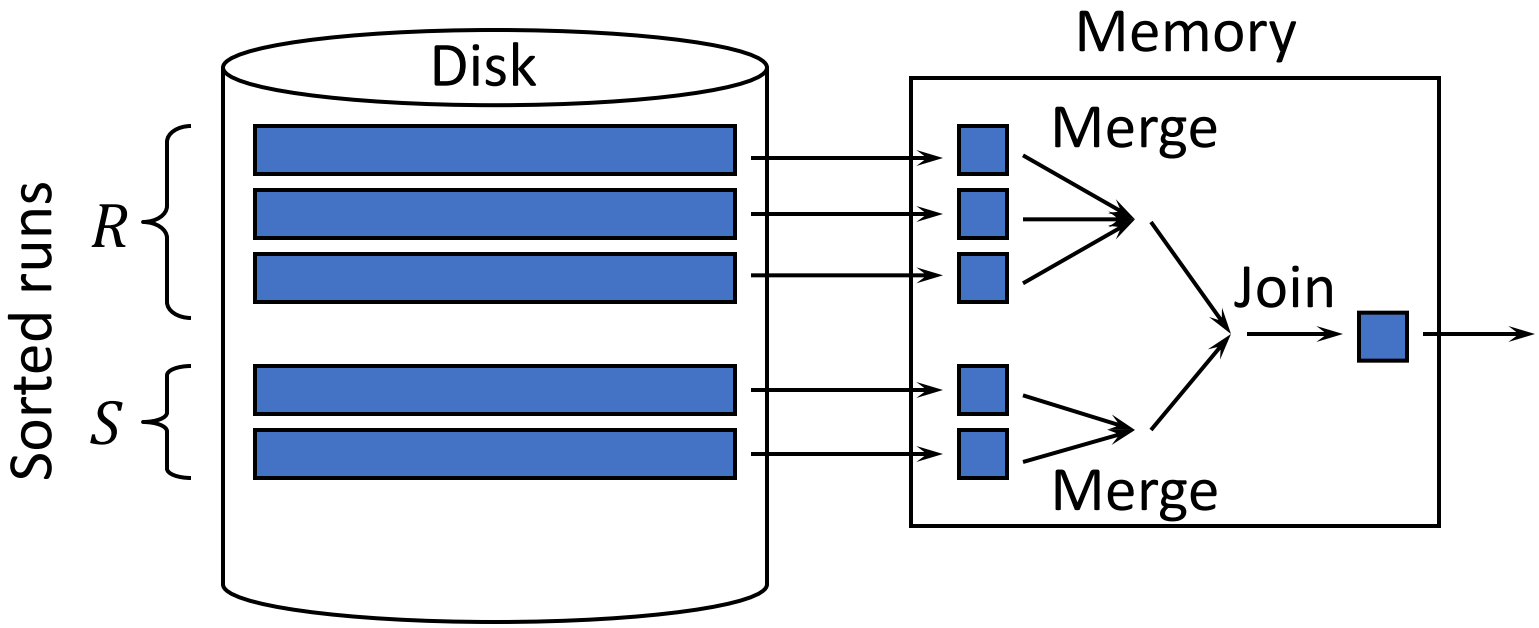
$r_3 s_3$

$r_3 s_4$

$r_7 s_5$

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: $3 \cdot \big(B(R) + B(S)\big)$
  - Memory requirement
    - We must have enough memory to accommodate one block from each run: $M > \left\lceil \frac{B(R)}{M} \right\rceil + \left\lceil \frac{B(S)}{M} \right\rceil$
    - Roughly $M > \sqrt{B(R) + B(S)}$
- 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

Overview

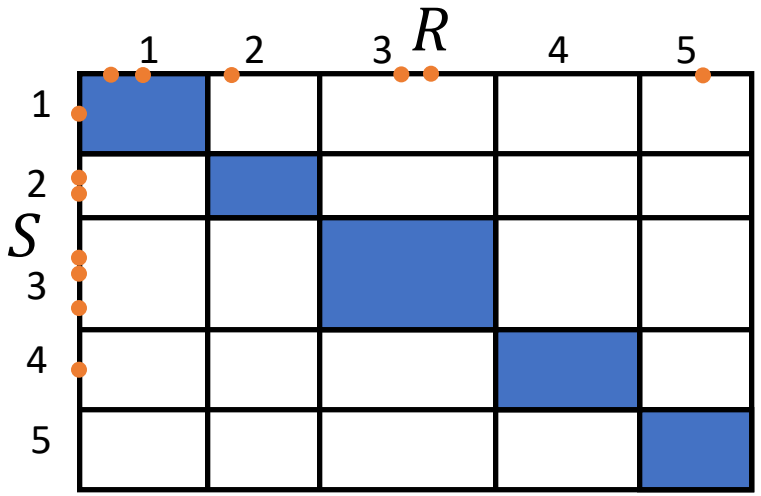
- Scanning-based algorithms
- Sorting-based algorithms
- **Hashing-based algorithms**
- Index-based algorithms

Hash join

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

Used for equality joins.
Not useful for inequality joins

- 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



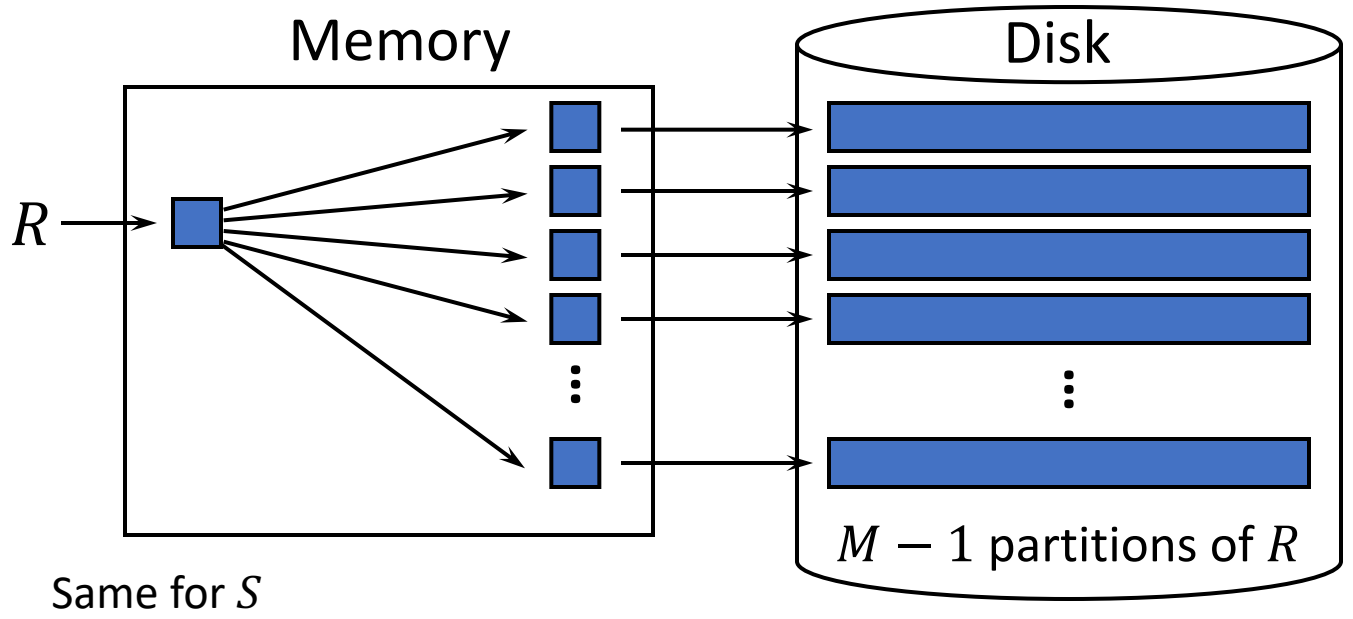
Nested-loop join considers all slots

Hash join considers only those along the diagonal!

Partitioning phase

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

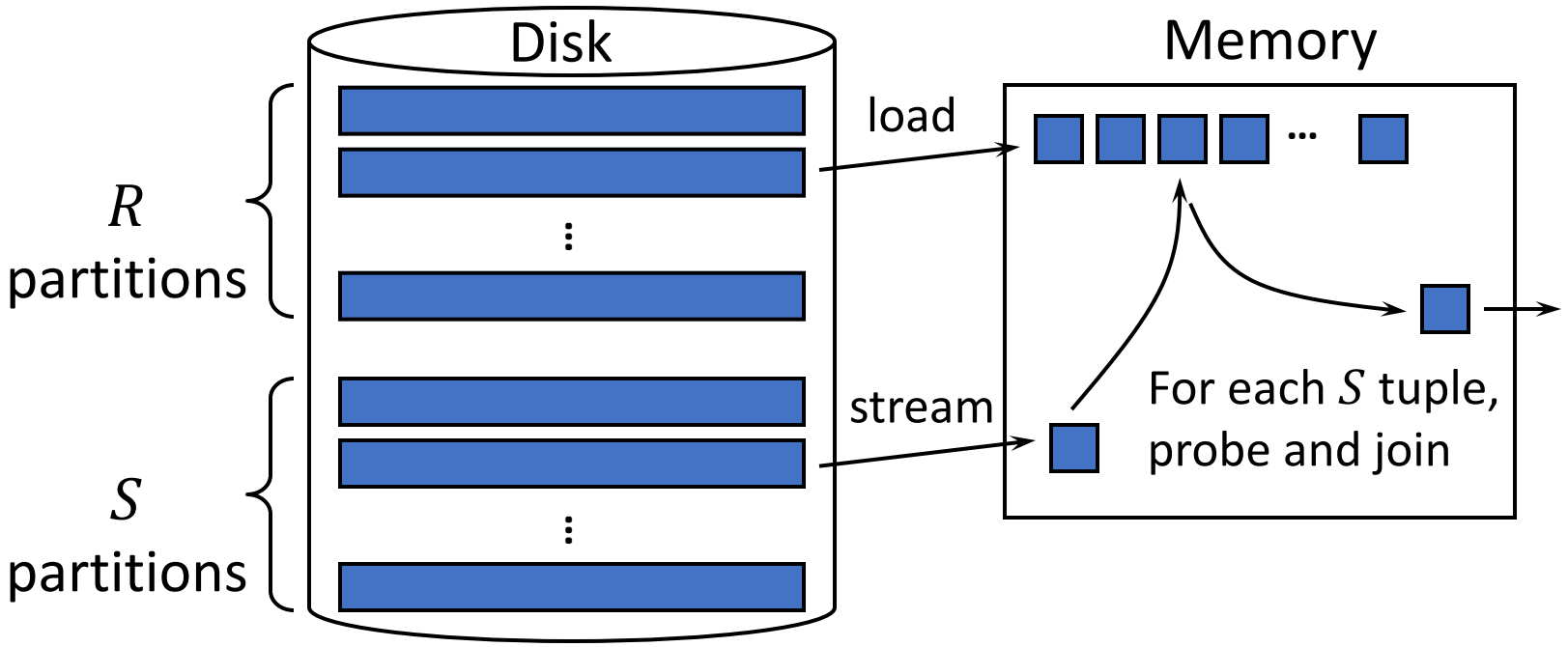
You need one memory block to act as a buffer, to transfer the data of R in and to write back the results.



Why are there M - 1 partitions of R?

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!



Performance of (two-pass) hash join

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

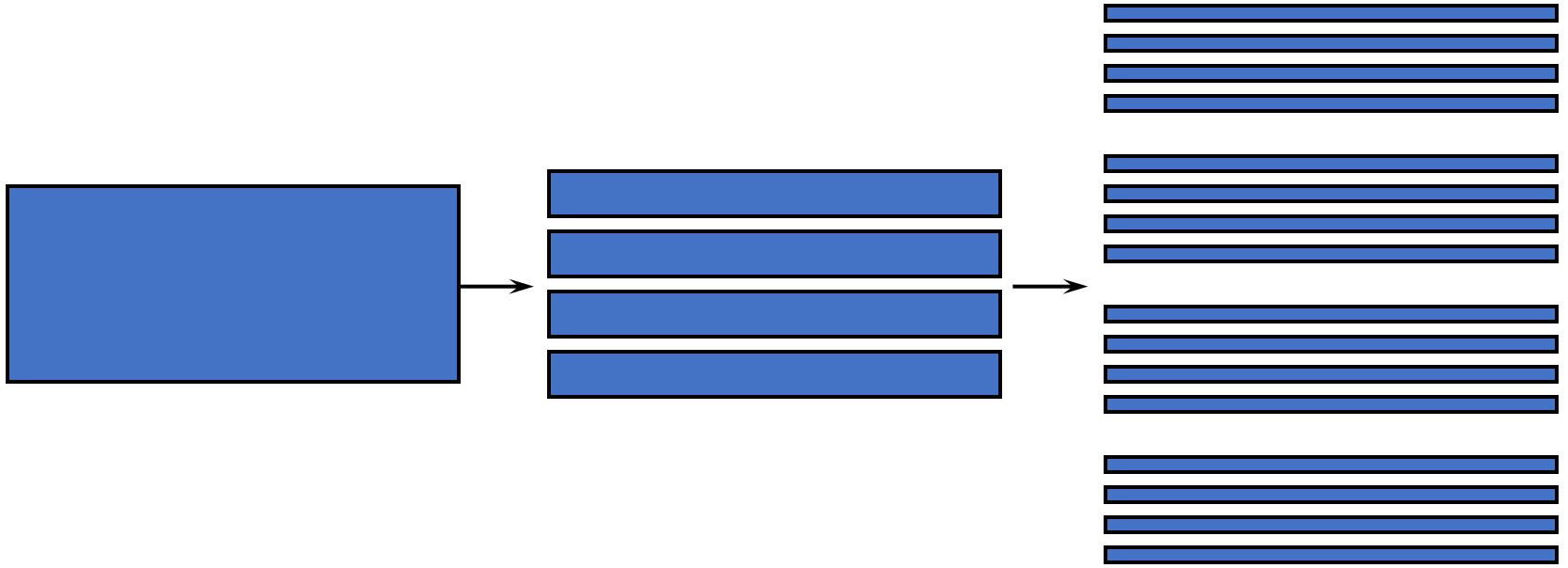
$$M > \sqrt{\min\big(B(R), B(S)\big)} + 1$$

Why are we multiplying by three?

You have to bring in all the data for the join. Then, you need to filter all the data and write back the results.
You also need enough space to store the intermediate results.
That is why you have 3 times the total number of blocks for R and S

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*($R.A, S.B$)

Other hash-based algorithms

- 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

Duality of sort and hash

- Divide-and-conquer paradigm
  - Sorting: physical division, logical combination
  - Hashing: logical division, physical combination
- Handling very large inputs
  - Sorting: multi-level merge
  - Hashing: recursive partitioning
- I/O patterns
  - Sorting: sequential write, random read (merge)
  - Hashing: random write, sequential read (partition)

Overview

- Scanning-based algorithms
- Sorting-based algorithms
- Hashing-based algorithms
- **Index-based algorithms**

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

Ask professor about the portion of the textbook you showed to the TA today

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

When using a primary key index, you can obtain the data without needing to go to disk again? Ask professor about this

Index versus table scan

Situations where index clearly wins:

- Index-only queries which do not require retrieving actual tuples
  - Example: $\pi_A(\sigma_{A>v}(R))$
- 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!

For each of the pointers, you will need to make a trip to disk.
Incurs a large cost.

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

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