Query Processing

Introduction to Databases

CompSci 316 Spring 2019

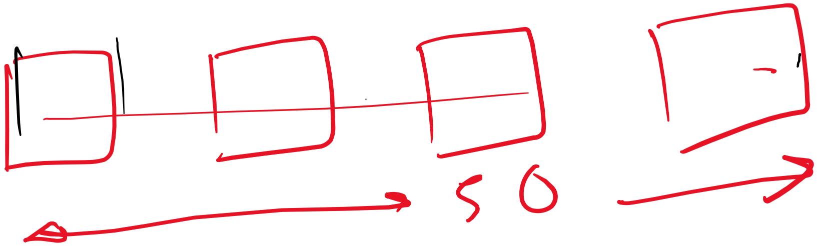
DUKE
COMPUTER SCIENCE

Announcements (Thu., Mar. 28)

- Project milestone #2 due this Friday
- Remember to submit project update on piazza by Friday

Overview

- Many different ways of processing 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

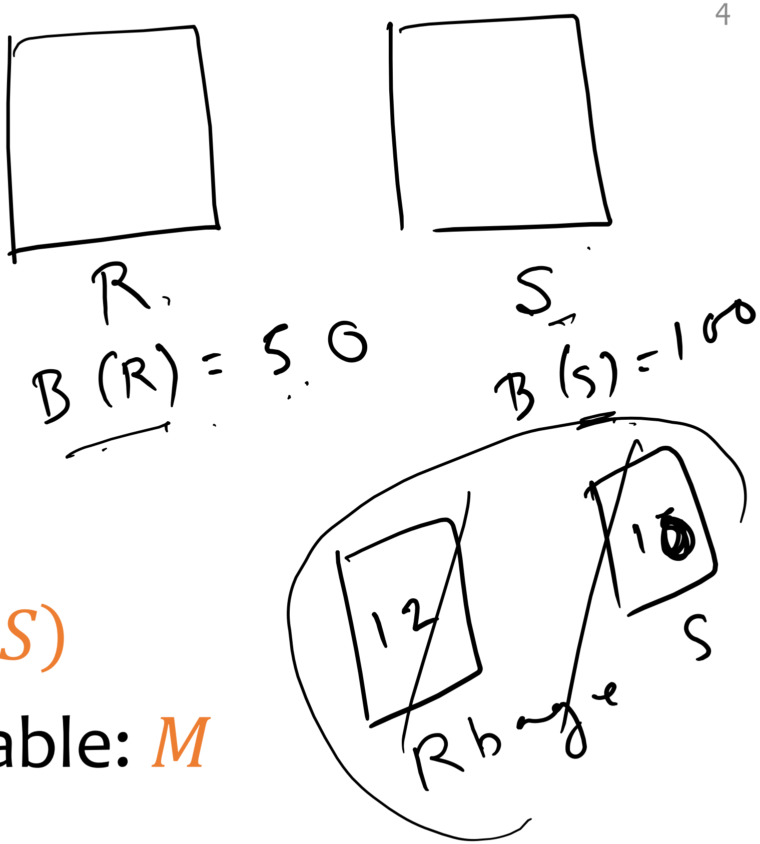


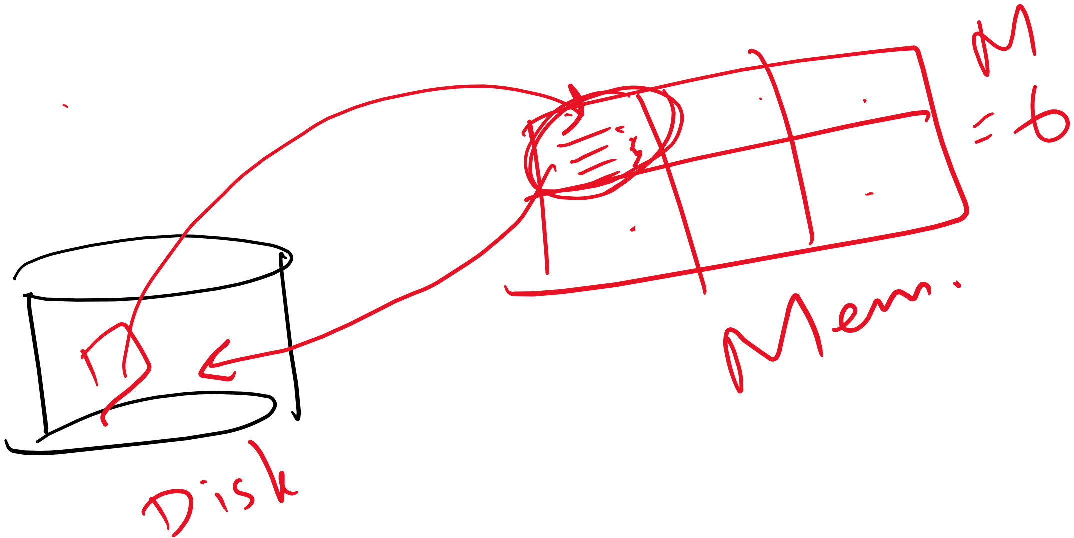
Notation

- 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

read: 1 1/6

write: 2 1/6





Scanning-based algorithms

Mem

2

R

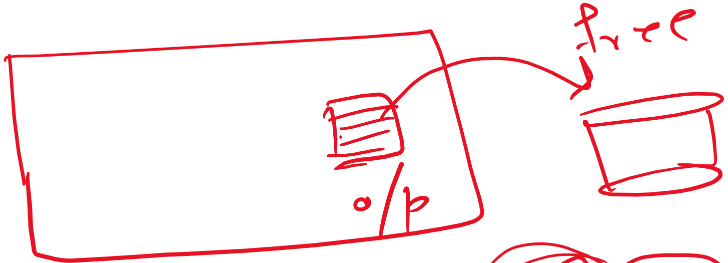
B(R)

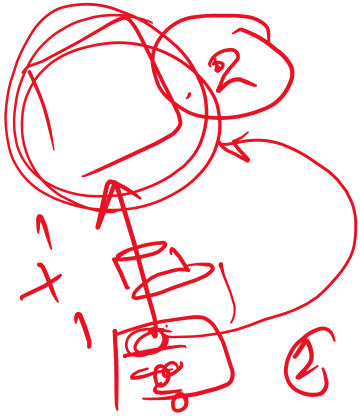
50

Select *
from R
where age = 15.

M = ?

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

Nested-loop join

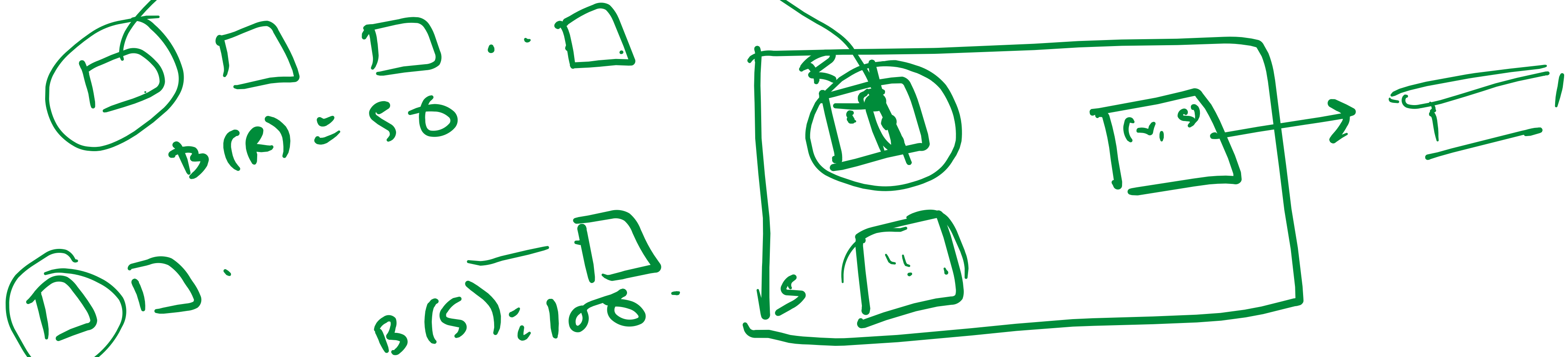
```
for all r ∈ R
    for all s ∈ S
        if r.A = s.B
            output (r, s)
```

R.A = S.B

AC $R \bowtie_p S$ BD

- 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

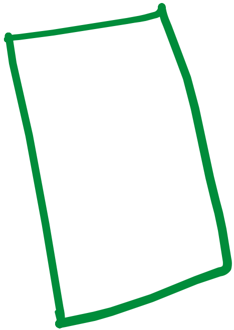
Improvement: block-based nested-loop join



Block-based Nested Loop Join

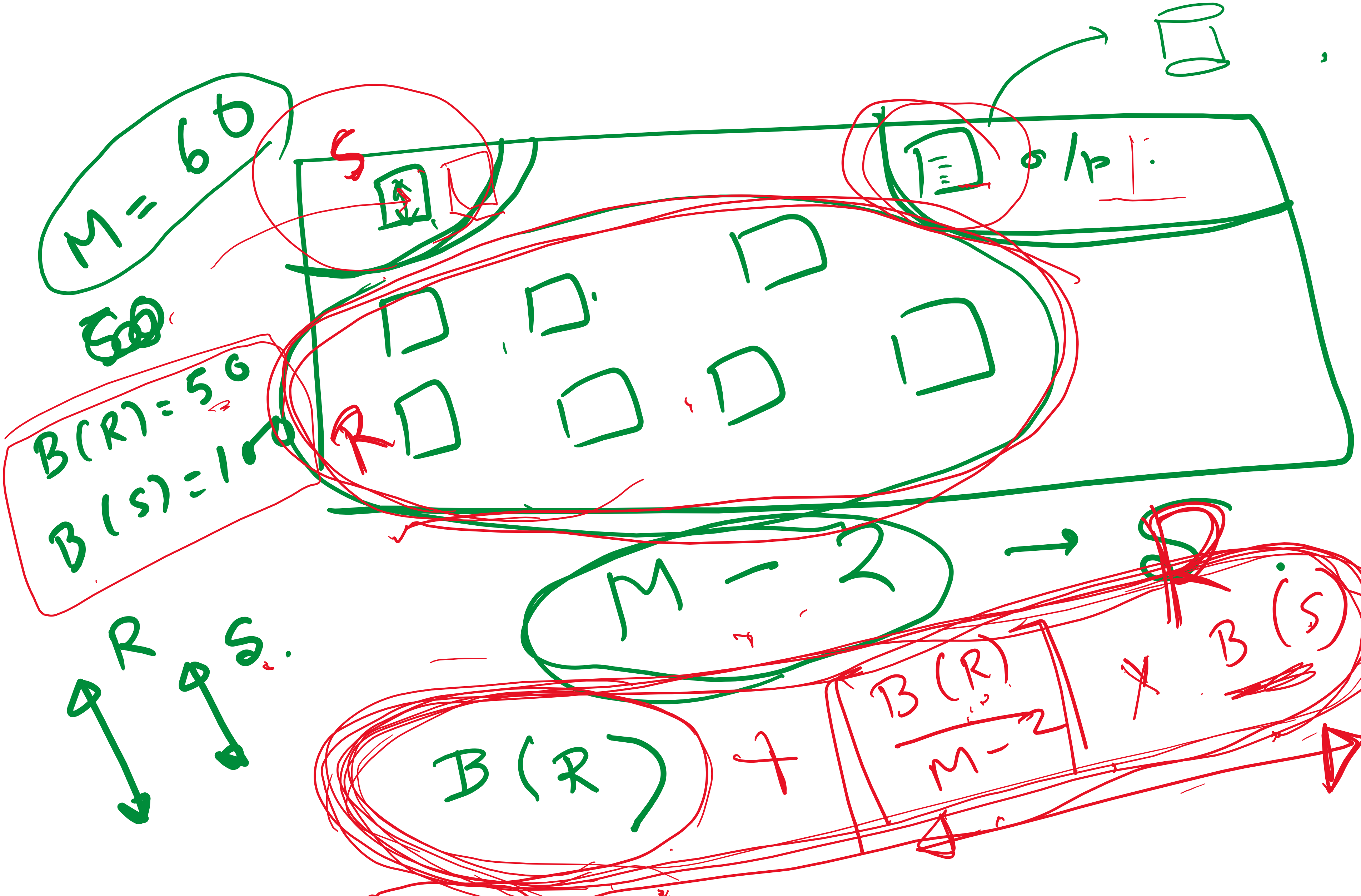
- $R \bowtie_p S$
- R outer, S inner
- For each block of R, for each block of S:
  For each r in the R block, for each s in the S block: …
  - I/O's: $B(R) + B(R) \cdot B(S)$
  - Memory requirement: same as before

$|R| = 50000$



1 page = 1000 R tuples

$B(R) = 50$


M = 60
B(R) = 56
B(S) = 100
S
R
o/p
M − 2
R
S
$B(R) + \boxed{\frac{B(R)}{M-2}} \times B(S)$

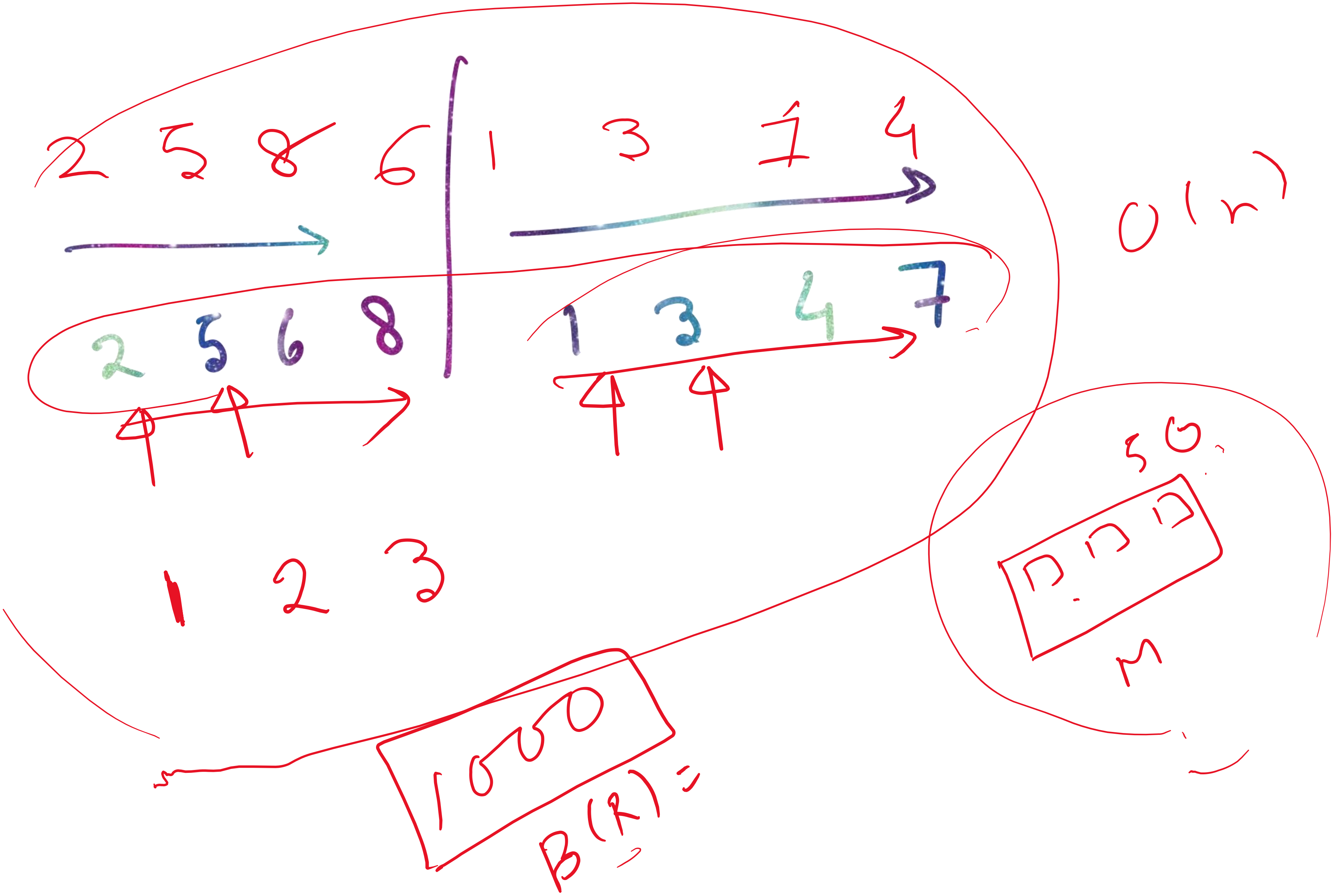
More improvements

- 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?

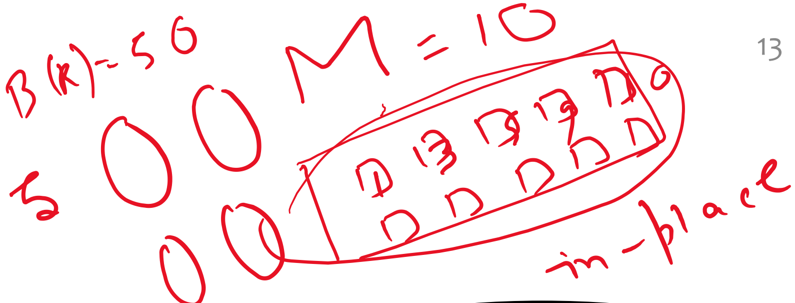
smaller

Sorting-based algorithms

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


2 5 8 6 | 1 3 7 4
2 5 6 8 | 1 3 4 7
1 2 3
O(n)
1000
B(R) =
5G
M

External merge sort

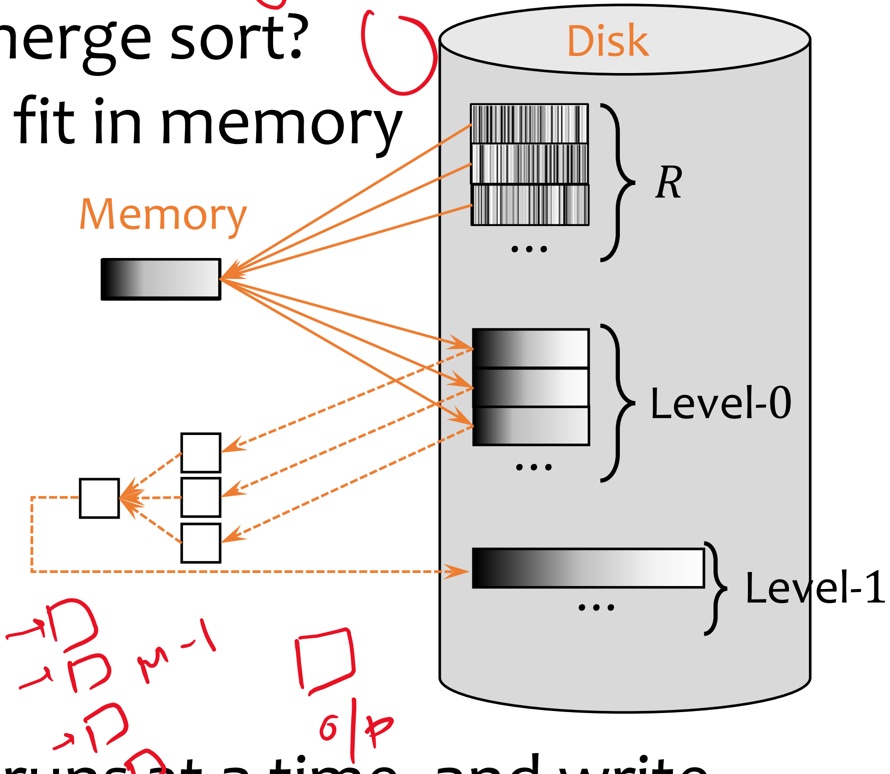


Remember (internal-memory) merge sort?

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

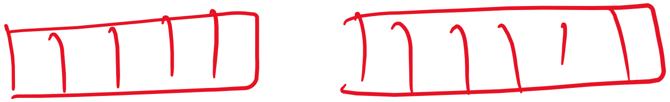


- Pass 2: merge $(M-1)$ level-1 runs at a time, and write out a level-2 run

…

- Final pass produces one sorted run

Toy example

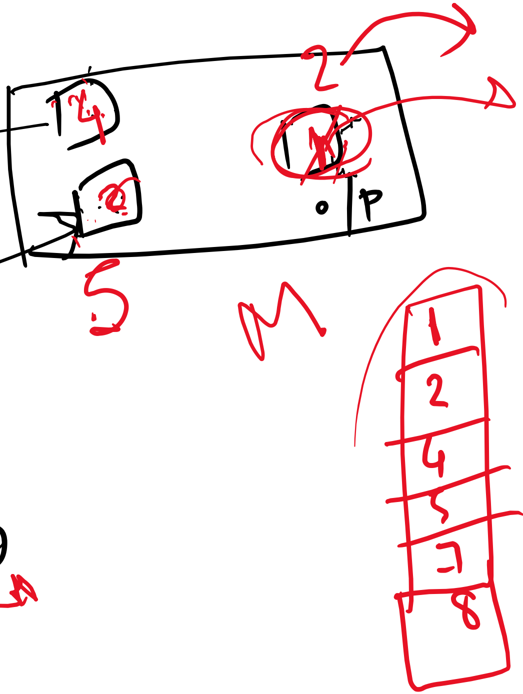


- 3 memory blocks available; each holds one number
- Input: 1, 7, 4, 5, 2, 8, 3, 6, 9   9, 6, 3
- Pass 0
  - 1, 7, 4 → 1, 4, 7   run = 3
  - 5, 2, 8 → 2, 5, 8
  - 9, 6, 3 → 3, 6, 9
- Pass 1
  - 1, 4, 7 + 2, 5, 8 → 1, 2, 4, 5, 7, 8
  - 3, 6, 9
- Pass 2 (final)
  - 1, 2, 4, 5, 7, 8 + 3, 6, 9 → 1, 2, 3,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

M = 3.

6

run =



M blocks.

R → B(R).

pass 0 → $\left\lceil \frac{B(R)}{M} \right\rceil = p_0$ runs. of size M

pass - 1 → (M-1) of them. of size. M * (M-1)

→ $\left\lceil \frac{p_0}{M-1} \right\rceil$

Analysis

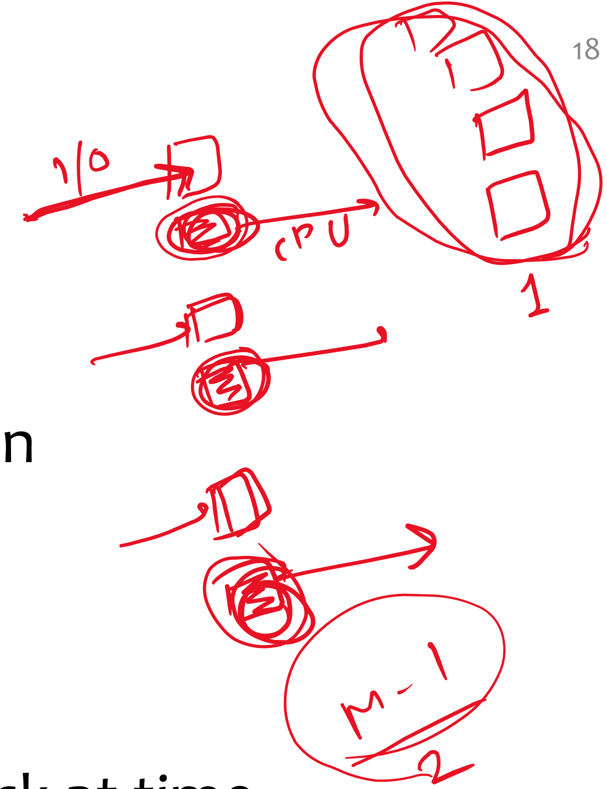
- Pass 0: read M blocks of R at a time, sort them, and write out a level-0 run
  - There are $\left\lceil \frac{B(R)}{M} \right\rceil$ level-0 sorted runs

    of
- 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: $\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil + 1$ # pass O
- 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)

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)

- Nested loop join – NLJ

- R \Join S

- For all r \in R
- For all s \in S
- Check if r and s join
- If yes, then output (r, s)

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: $\text{sorting} + 2B(R) + 2B(S)$ (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:

➡ $s_1.B = 1$

➡ $s_2.B = 2$

➡ $s_3.B = 3$

$s_4.B = 3$

➡ $s_5.B = 8$

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

$r_1 s_1$

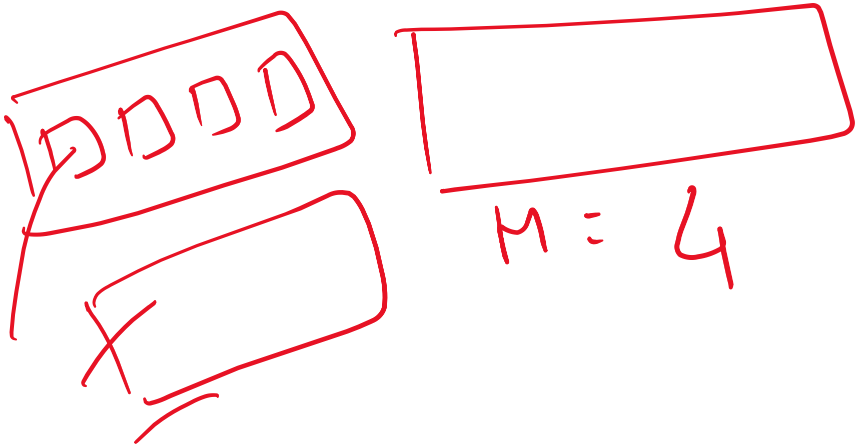
$r_2 s_3$

$r_2 s_4$

$r_3 s_3$

$r_3 s_4$

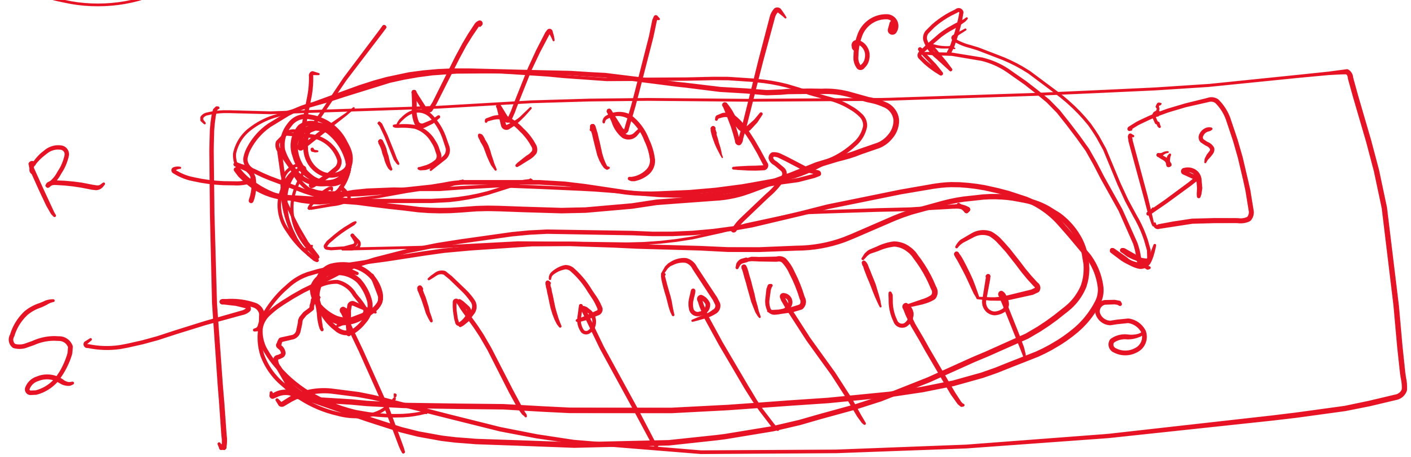
$r_7 s_5$



36

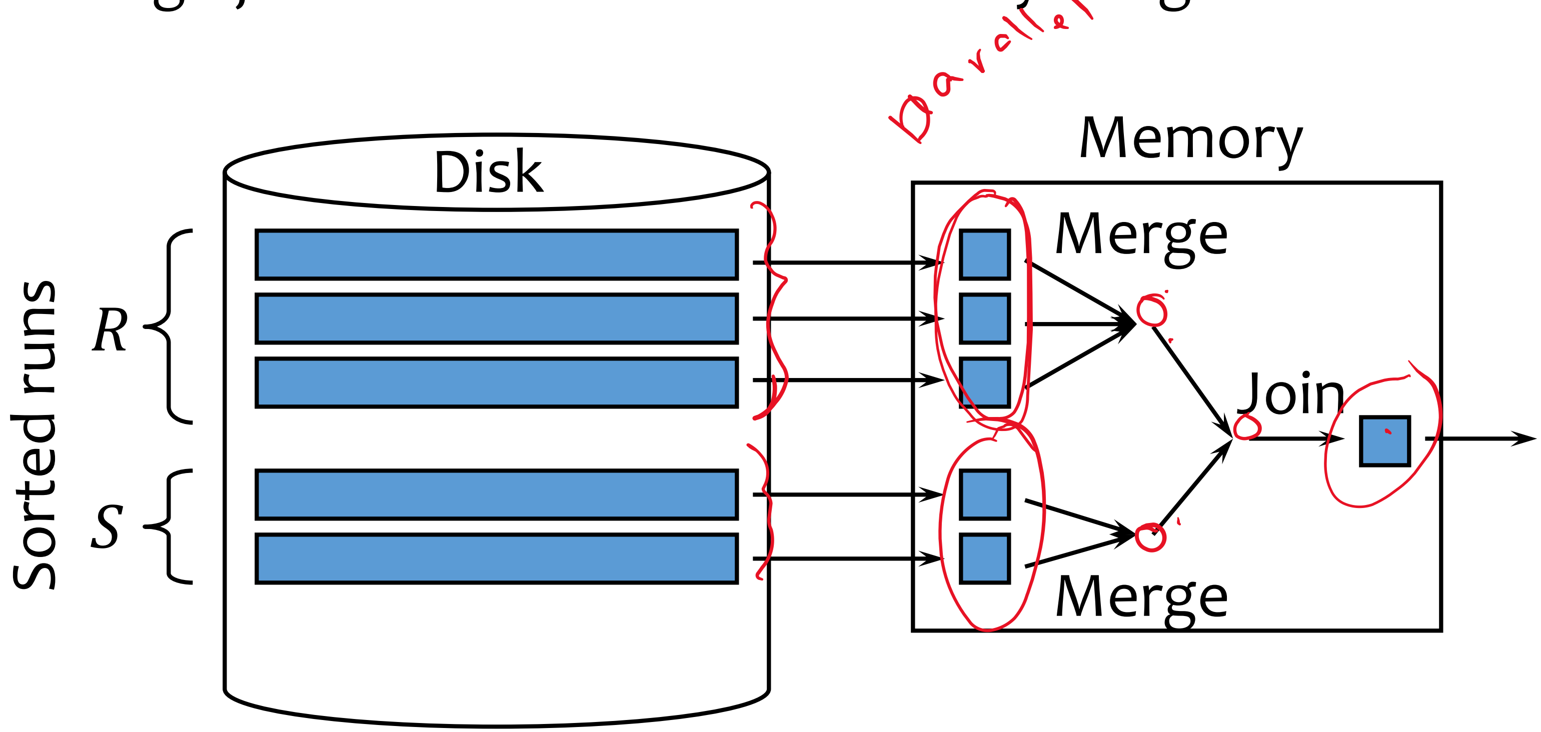
$\left\lceil \frac{50}{4} \right\rceil$ runs.

3-way



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

pass 0

1. so make runs of R
make runs of S
2 make runs R S
3 join

- 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: $M > \frac{B(R)}{M} + \frac{B(S)}{M}$
    - $M > \sqrt{B(R) + B(S)}$

$M \; \boxed{+1} = $ of,

- If SMJ cannot complete in two passes:
  - Repeatedly merge to reduce the number of runs as necessary before final merge and join

$B(s) = 70$

$\left\lceil \frac{70}{4} \right\rceil$ $\boxed{18}$ - way

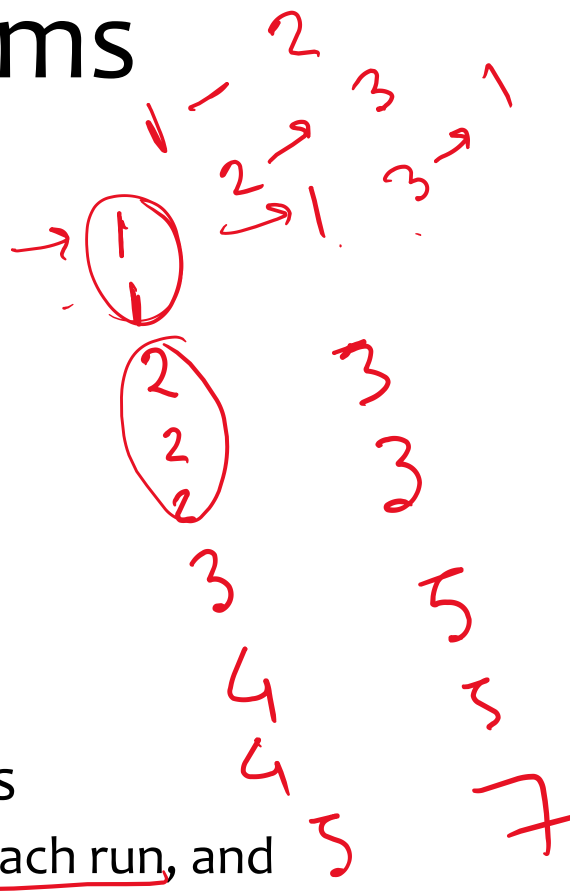
pass 0

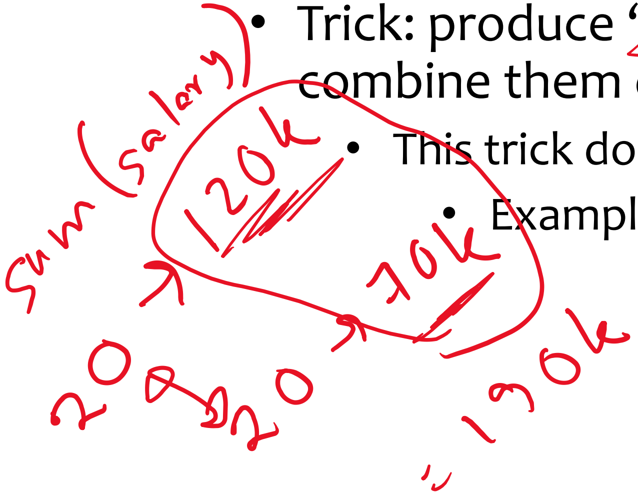
$B(R) = 90$

$\left\lceil \frac{90}{4} \right\rceil$ $\boxed{23}$ - way.

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(...)

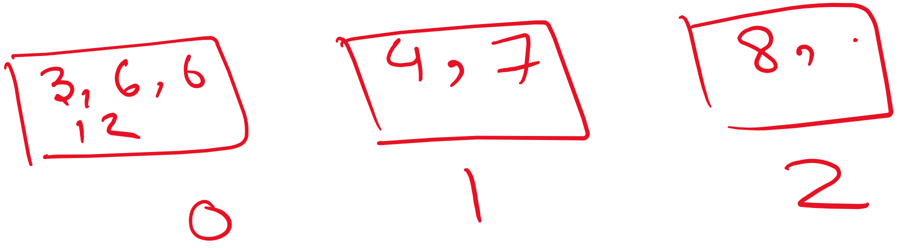




Hashing-based algorithms

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

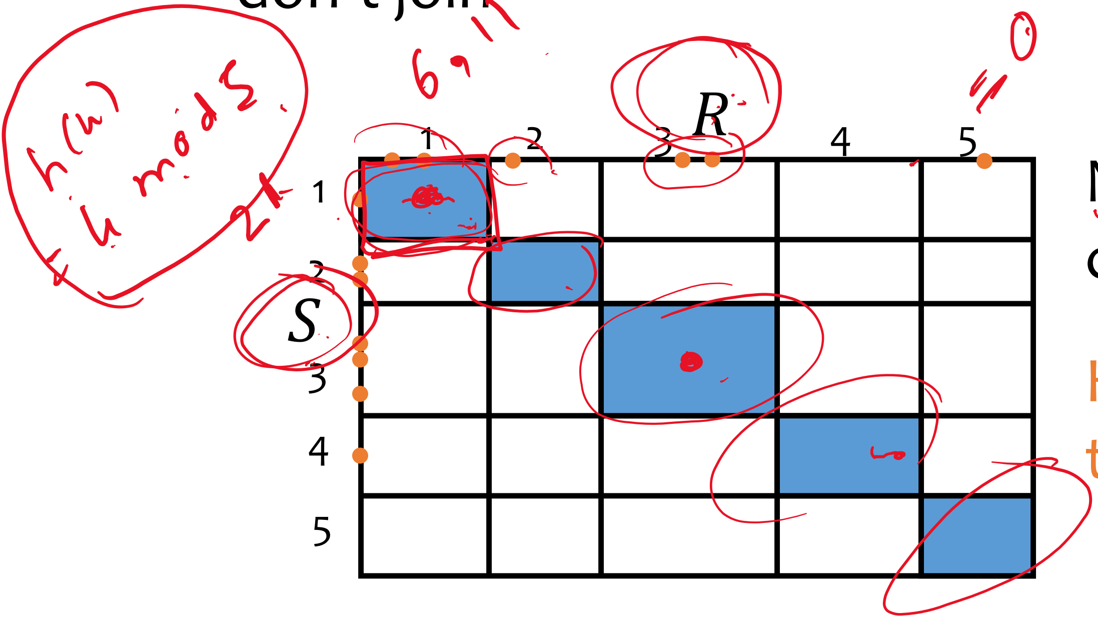
$h(k) = k \bmod 3$



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

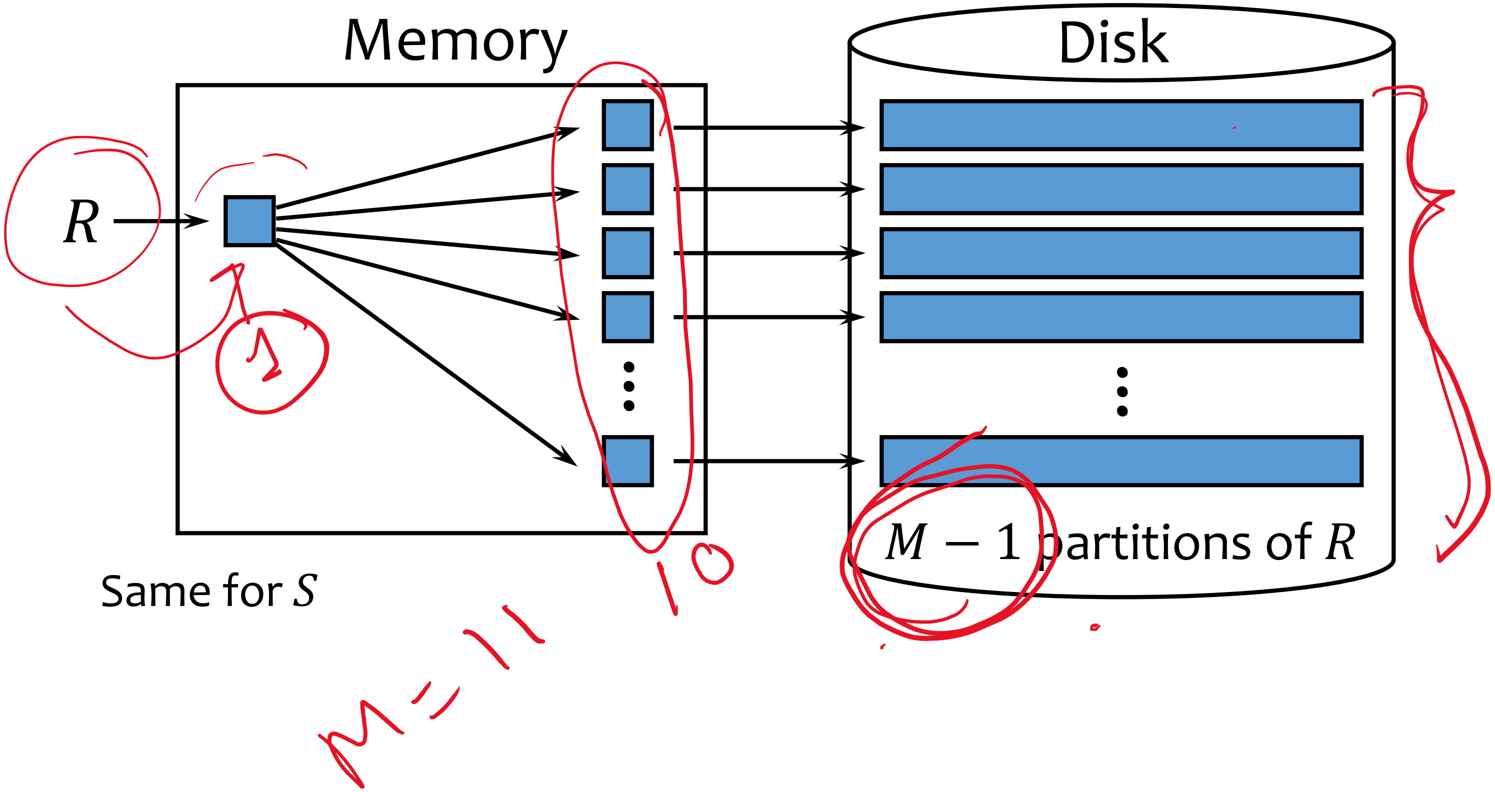
Hash join considers only those along the diagonal!

1

Partitioning phase

$h'(k) = h(k) \bmod 10$

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



$M-1$ partitions of R

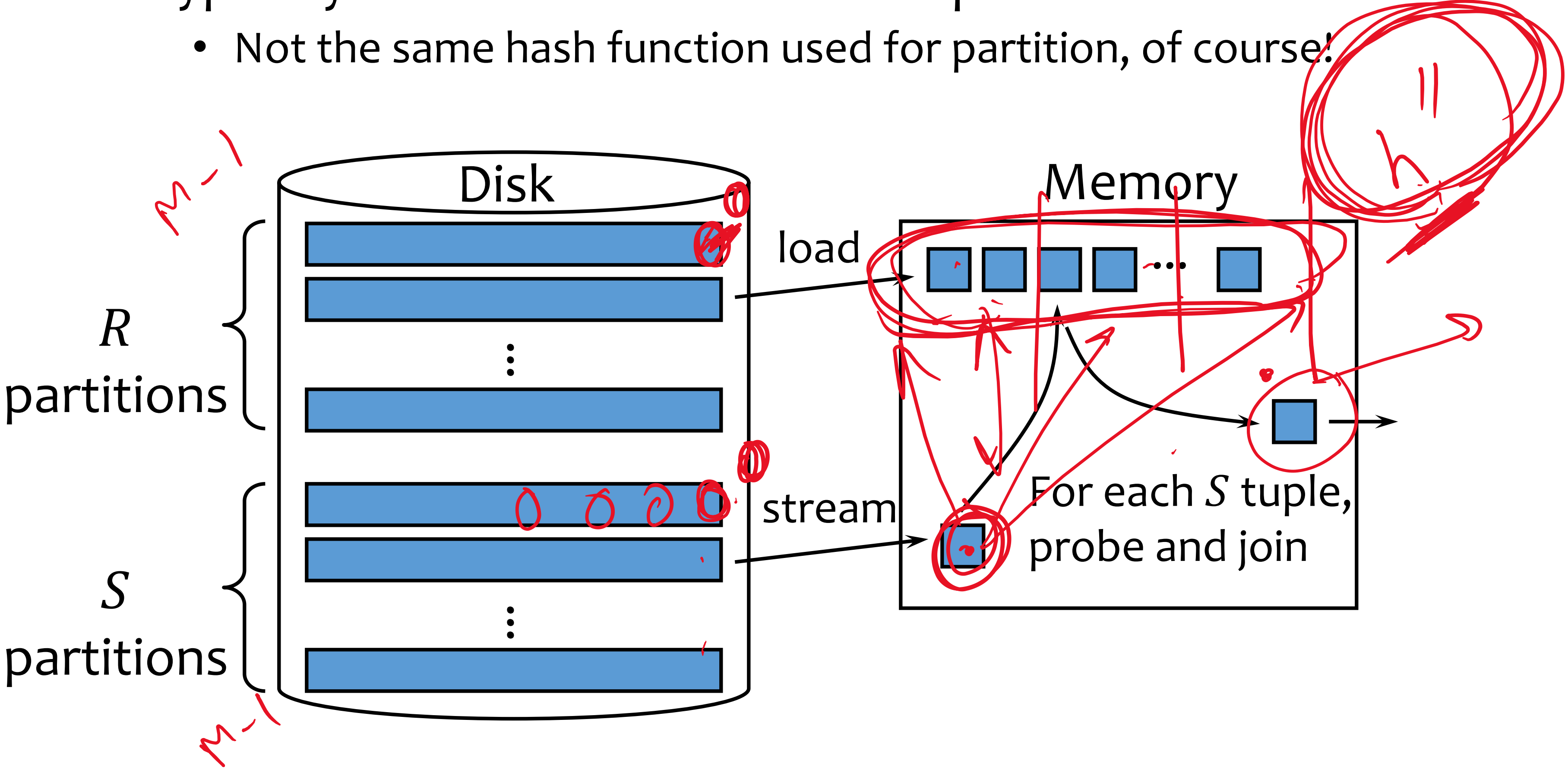
Same for S

2

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!

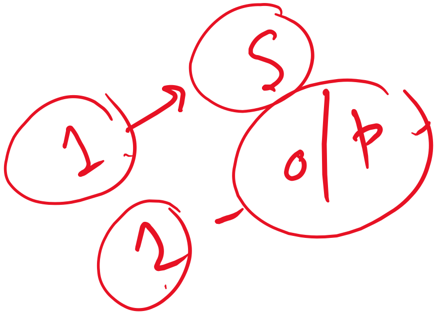
h



Performance of (two-pass) hash join

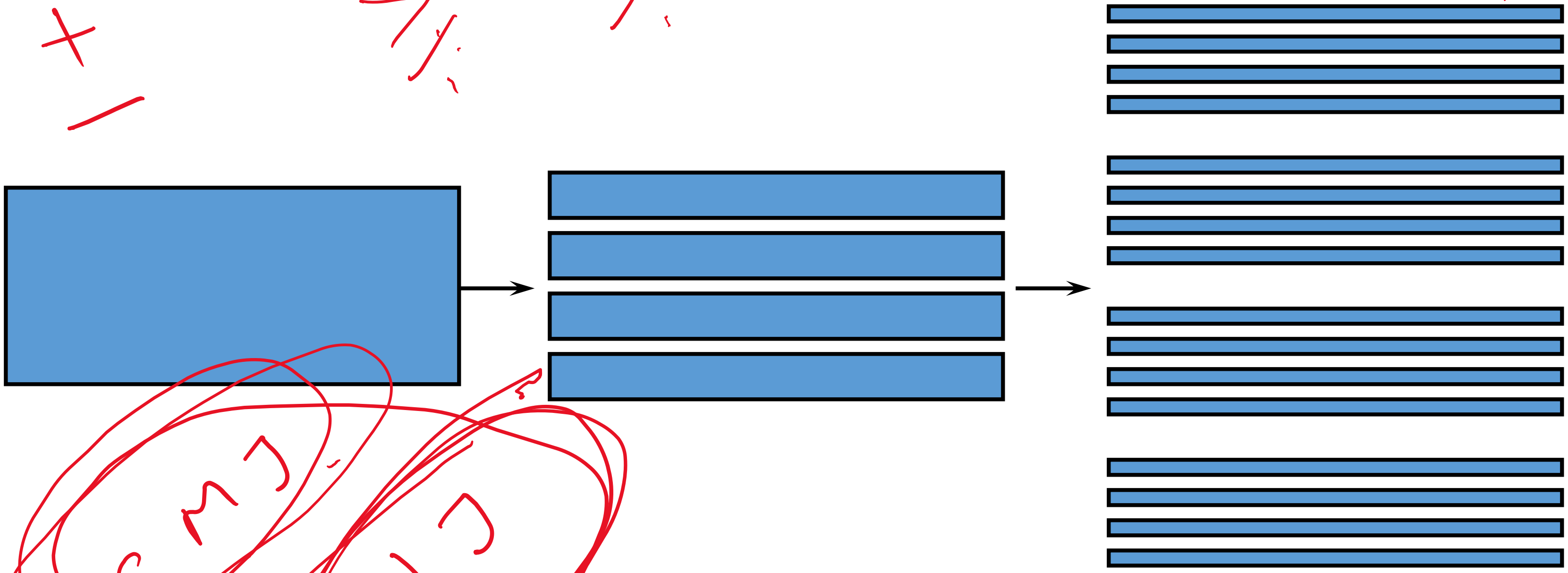
- If hash join completes in two passes:
  - I/O's: $3 \cdot \left(B(R) + B(S)\right)$
  - Memory requirement:
    - In the probing phase, we should have enough memory to fit one partition of R: $M - 1 > \frac{B(R)}{M-1}$ = size of 1 partition
    - $M > \sqrt{B(R)} + 1$
    - We can always pick R to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

3 × (BR + BS)

ORDER BY

- 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?

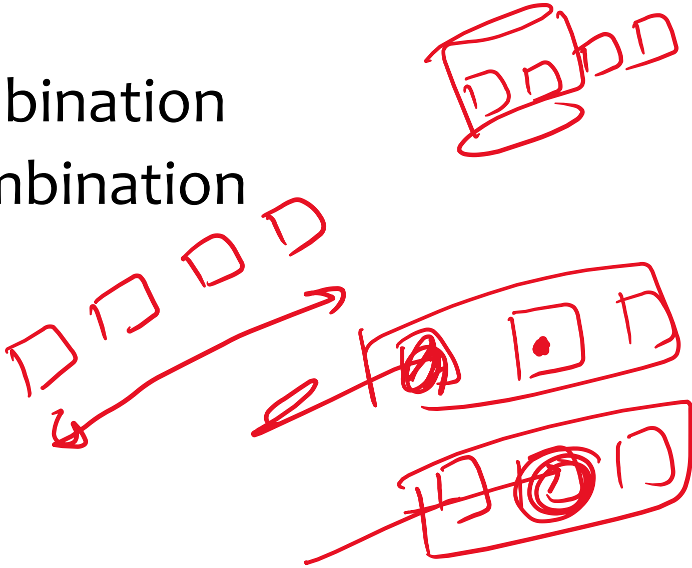
- May be best if many tuples join
  - Example: non-equality joins that are not very selective
- Necessary for black-box predicates
  - Example: WHERE $\mathit{user\_defined\_pred}(R.A, S.B)$

Other hash-based algorithms

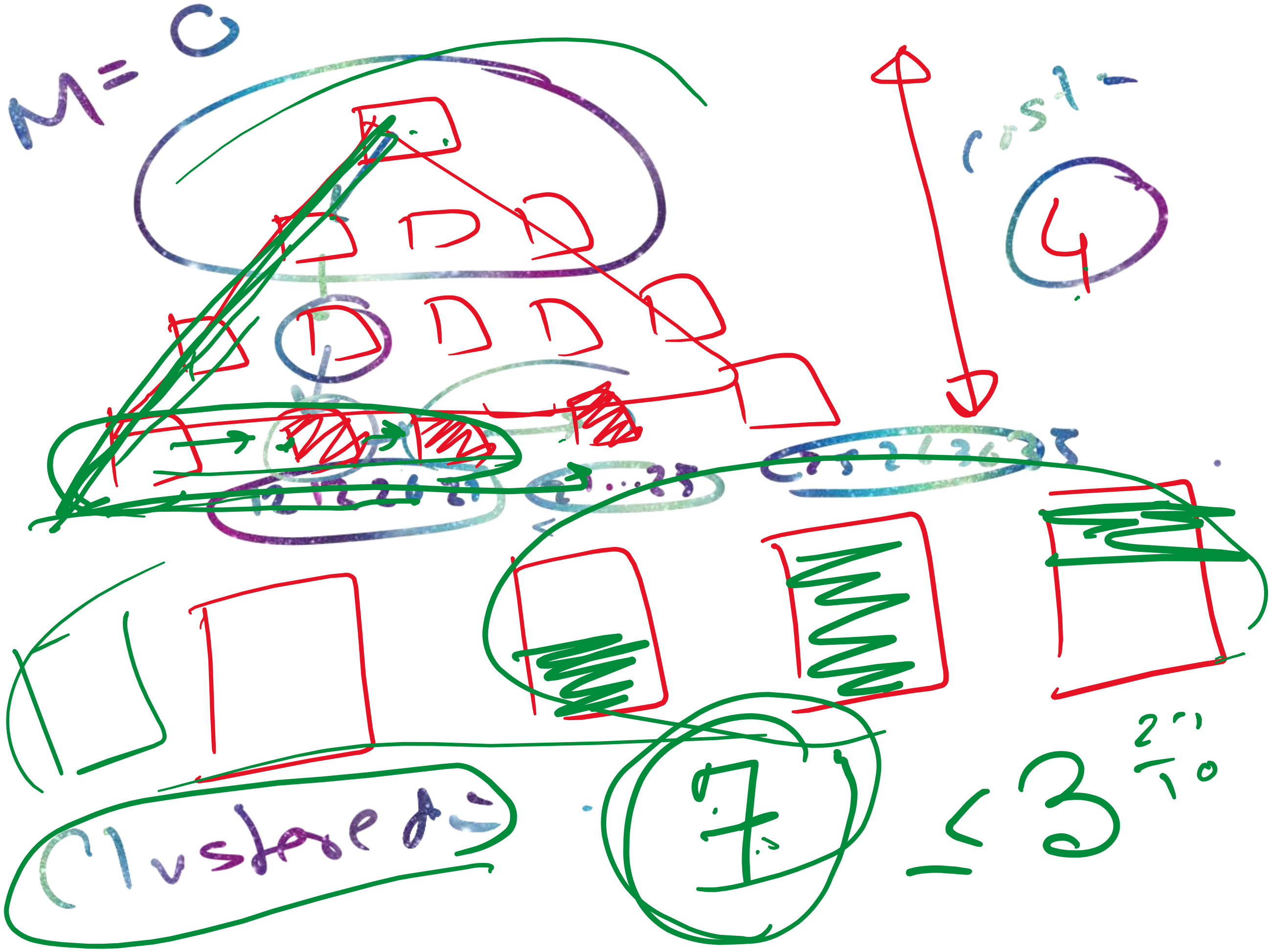
- Union (set), difference, intersection
  - More or less like hash join
- Duplicate elimination
  - Check for duplicates within each partition/bucket
- Grouping and aggregation
  - Apply the hash functions to the group-by columns
  - Tuples in the same group must end up in the same partition/bucket
  - 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)



select *

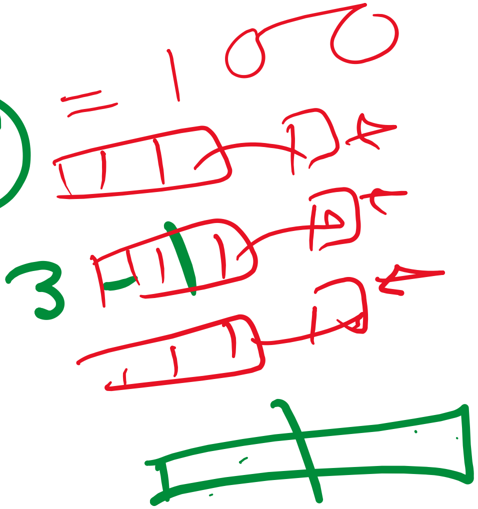


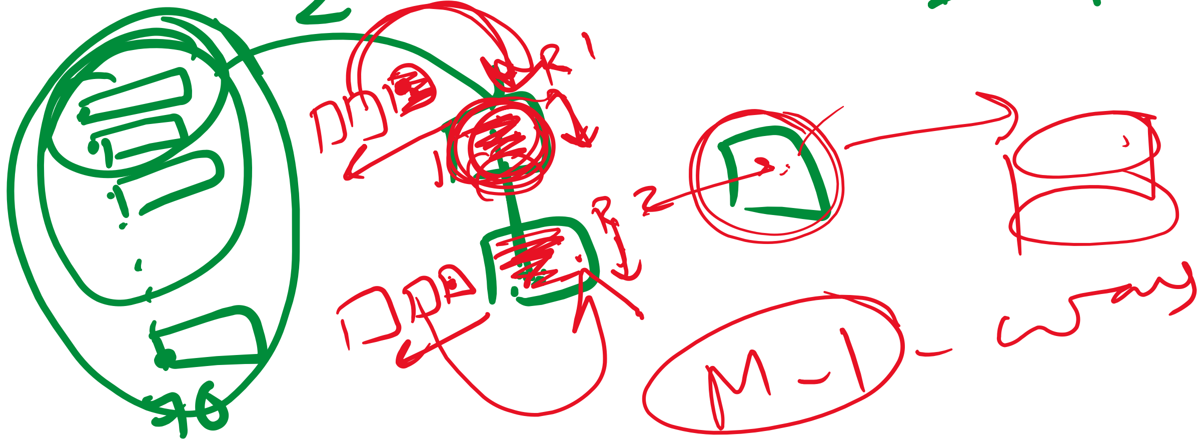
$M = 3$

$\lceil \frac{B}{M} \rceil = 70$

2-way

M = 100





M-1 - way

$B(R) = 200 \left\lceil \frac{13}{4} \right\rceil$ 3r 4

1 - 1

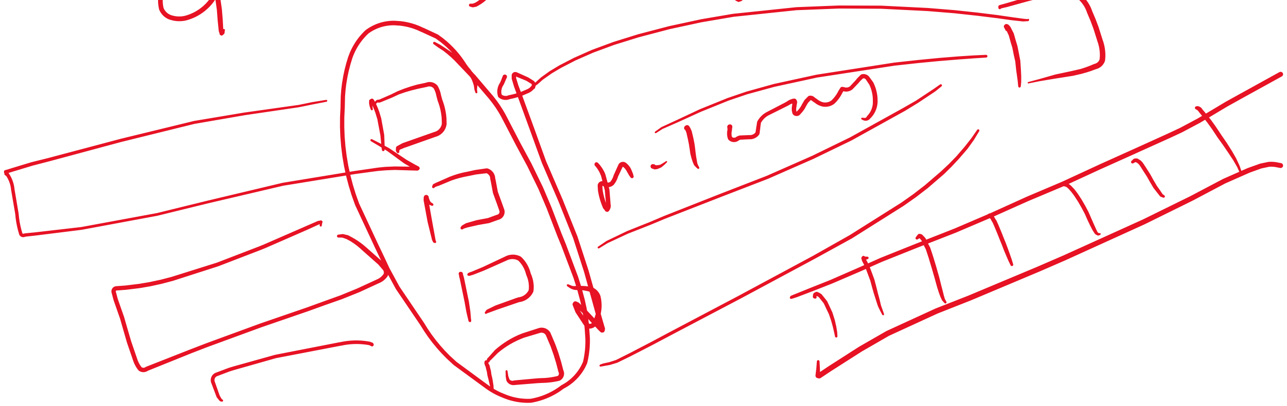
$M = 5$

pass 0 → $\frac{200}{5} = 40$

5x

4-way

$5 * 4 = 20$



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:

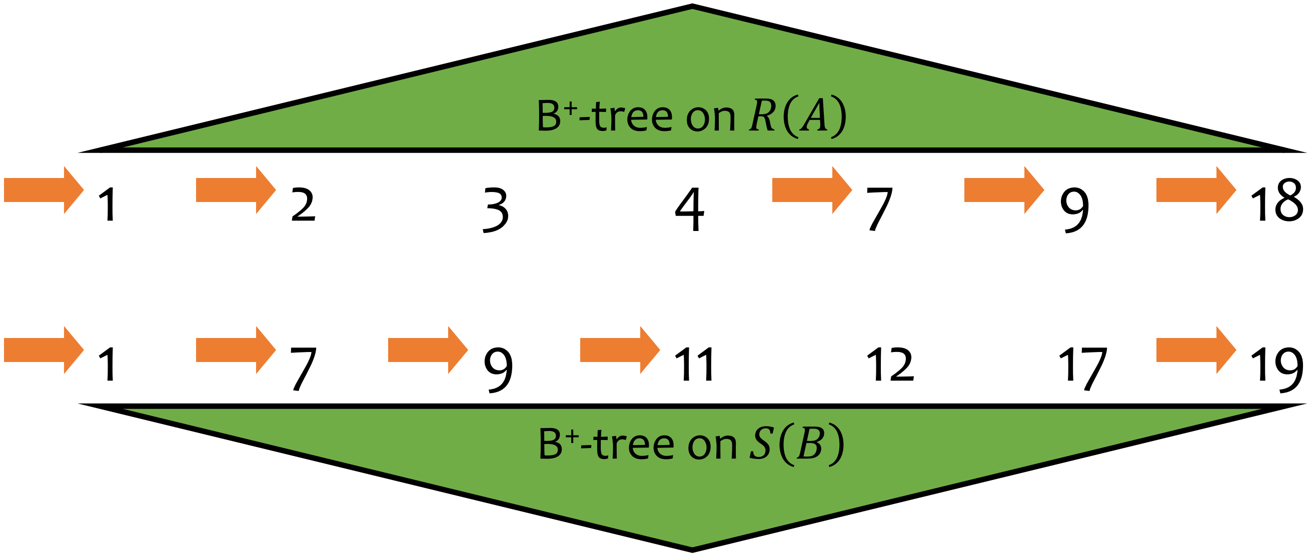
```
    Use the index on $S(B)$ to retrieve $s$ with $s.B = r.A$
        Output $rs$
```

- 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

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