# Query Processing Sort/Hash-based (Optional)

Introduction to Database Management

CS348 Fall 2022

# Outline

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

# Sorting-based algorithms

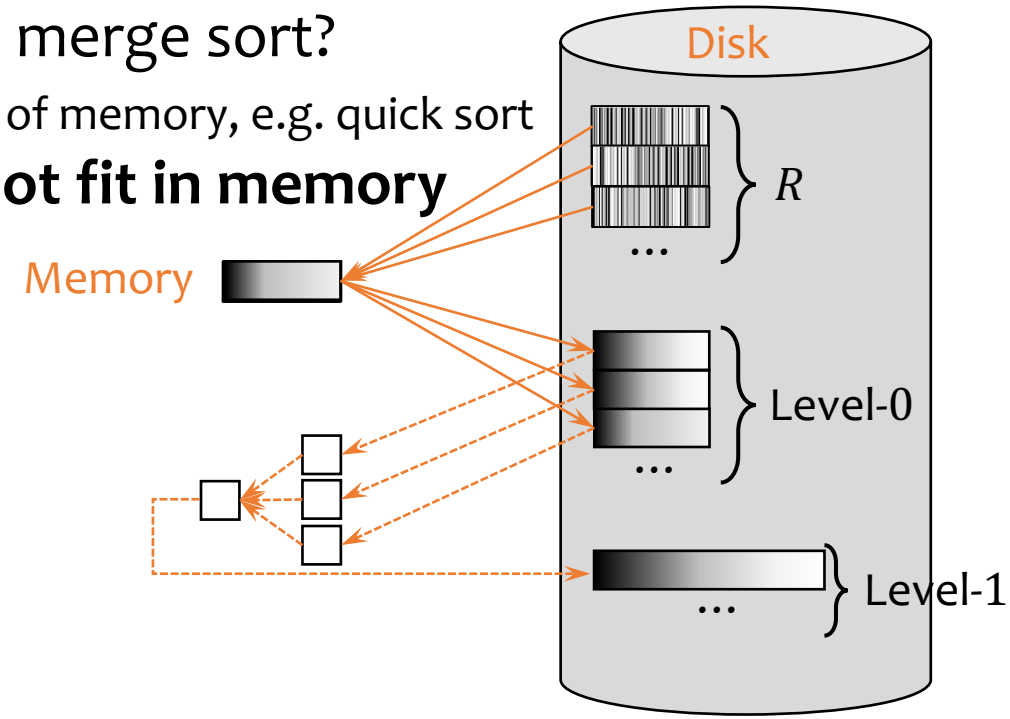
http://en.wikipedia.org/wiki/Mail_sorter#mediaviewer/File:Mail_sorting,1951.jpg

# External merge sort

Remember (internal-memory) merge sort?

-- sort $M$ blocks of data with $M$ blocks of memory, e.g. quick sort

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




- Phase 0: read $M$ blocks of $R$ at a time, sort them, and write out a level-0 run
- Phase 1: merge $(M-1)$ level-0 runs at a time, and write out a level-1 run
- Phase 2: merge $(M-1)$ level-1 runs at a time, and write out a level-2 run

…

- Final phase produces one sorted run

# Toy example

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

# Analysis

- Phase 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

I/O cost is $2 \cdot B(R)$

- Phase $i$: 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
  - The number of level-$i$ runs = $\left\lceil \frac{number \text{ of level}-(i-1) \text{ runs}}{M-1} \right\rceil$
  - $\left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil$ number of such phases
  - Final pass produces one sorted run

I/O cost is $2 \cdot B(R)$ times # of phases

Subtract $B(R)$ for the final pass

# Performance of external merge sort

- I/O's
  - $2\mathrm{B(R)} \cdot \left(1 + \left\lceil \log_{M-1} \left\lceil \frac{B(R)}{M} \right\rceil \right\rceil\right) - \mathrm{B(R)}$
  - Roughly, this is $O\left(B(R) \times \log_M B(R)\right)$

- Memory requirement: $M$ (as much as possible)

# Case study:

- System requirements:
  - Each disk/memory block can hold up to 10 rows (from any table);
  - All tables are stored compactly on disk (10 rows per block);
  - 8 memory blocks are available for query processing: M=8
- Database:
  - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
  - |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
  - #of blocks: B(User)=1000/10=100; B(Group)=100/10=10; B(Member)=50000/10=5k
- Q3: select * from User order by age asc;
  - I/O cost using external merge sort?

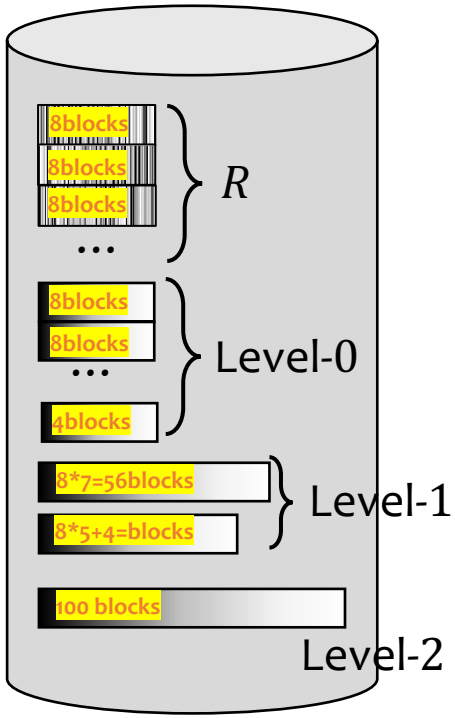
# Case study:

- System requirements:
  - Each disk/memory block can hold up to 10 rows (from any table);
  - All tables are stored compactly on disk (10 rows per block);
  - 8 memory blocks are available for query processing: M=8
- Database:
  - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
  - |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
  - #of blocks: B(User)=1000/10=100; B(Group)=100/10=10; B(Member)=50000/10=5k
- Q3: select * from User order by age asc;
  - I/O cost using external merge sort?
    - Phase 0: read 8 blocks into memory at a time and sort it => ceil(100/8)=13 runs
    - Phase 1: merge 7 runs at a time => ceil(13/7)=2 runs
    - Phase 2: merge last 2 runs into a single run

*Number of phases:* $\left\lceil \log_{M-1} \left\lceil \frac{B(User)}{M} \right\rceil \right\rceil + 1 = \left\lceil \log_{(8-1)} \left\lceil \frac{100}{8} \right\rceil \right\rceil + 1 = 3$

*Phase 0: read B(user)=100 blocks, write B(User)=100 blocks (temporary result)*
*Phase 1: read B(user)=100 blocks, write B(User)=100 blocks (temporary result)*
*Phase 2: read B(user)=100 blocks, write B(User)=100 blocks (final result, don't count)*

*Total:* $2B(User) * 3 - B(User) = 5B(user) = 500$



# 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 $+O(B(R) + B(S))$
  - 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

$R$:

- $r_1.A = 1$
- $r_2.A = 3$
- $r_3.A = 3$
- $r_4.A = 5$
- $r_5.A = 7$
- $r_6.A = 7$
- $r_7.A = 8$

$S$:

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

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

$r_1 s_1$

$r_2 s_3$

$r_2 s_4$

$r_3 s_3$

$r_3 s_4$

$r_7 s_5$

# Case study:

- System requirements:
  - Each disk/memory block can hold up to 10 rows (from any table);
  - All tables are stored compactly on disk (10 rows per block);
  - 8 memory blocks are available for query processing: M=8
- Database:
  - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
  - |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
  - #of blocks: B(User)=1000/10=100; B(Group)=100/10=10; B(Member)=50000/10=5k
- Q2: select * from User, Member where User.uid = Member.uid;
  - I/O cost using SMJ?
  - Sorting cost for User:
    (assume uid unsorted yet)
  - Sorting cost for Member:
    (assume uid unsorted)
  - Join cost: foreign-key and primary key join

# Case study:

- System requirements:
  - Each disk/memory block can hold up to 10 rows (from any table);
  - All tables are stored compactly on disk (10 rows per block);
  - 8 memory blocks are available for query processing: M=8
- Database:
  - User(uid, age, pop), Member(gid,uid,date), Group(gid, gname)
  - |User|=1000 rows, |Group|=100 rows, |Member|=50000 rows
  - #of blocks: B(User)=1000/10=100; B(Group)=100/10=10; B(Member)=50000/10=5k
- Q2: select * from User, Member where User.uid = Member.uid;
  - I/O cost using SMJ?
  - Sorting cost for User: (assume uid unsorted yet)

$$\#of\ phases \left\lceil \log_{M-1} \left\lceil \frac{B(User)}{M} \right\rceil \right\rceil + 1 = \left\lceil \log_{(8-1)} \left\lceil \frac{100}{8} \right\rceil \right\rceil + 1 = 3$$

$$2B(User) * 3 - B(User) = 5B(user) = 500$$

  - Sorting cost for Member: (assume uid unsorted)

$$\#of\ phases \left\lceil \log_{M-1} \left\lceil \frac{B(Member)}{M} \right\rceil \right\rceil + 1 = \left\lceil \log_{(8-1)} \left\lceil \frac{5K}{8} \right\rceil \right\rceil + 1 = 5$$

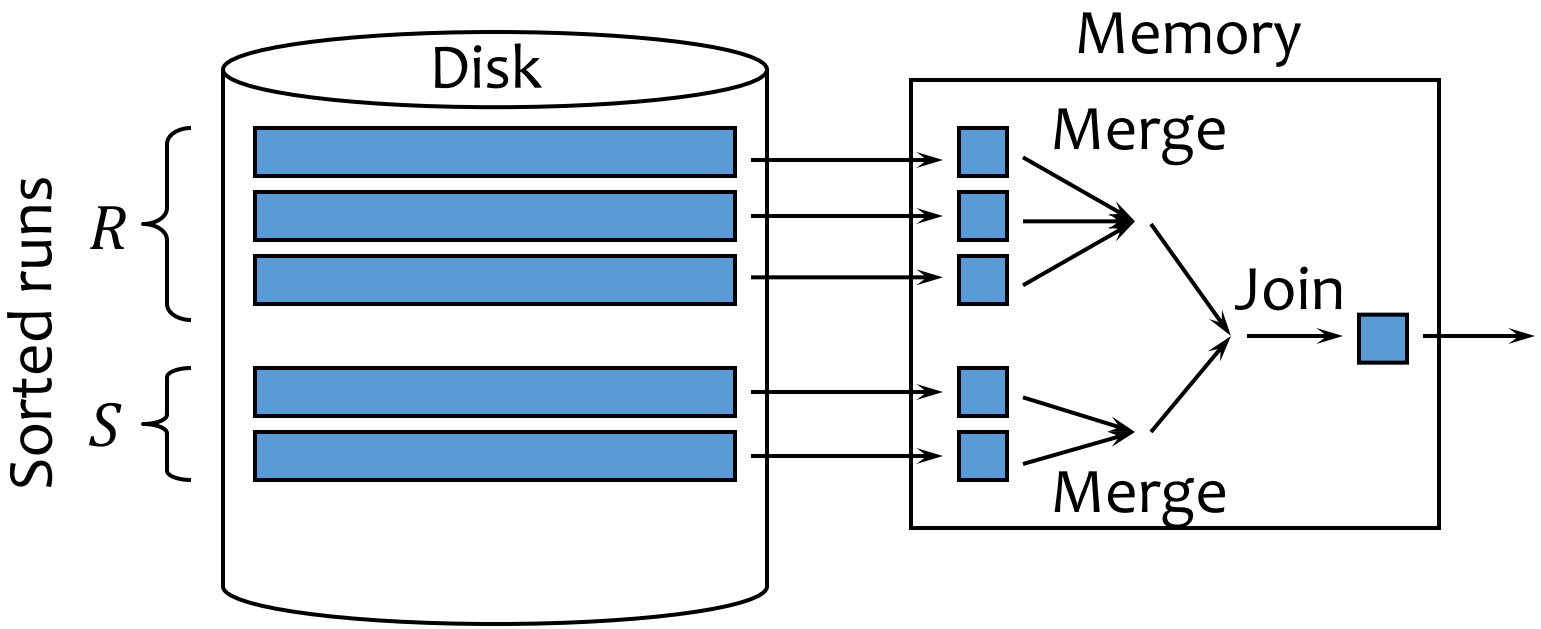
$$2B(Member) * 5 - B(Member) = 9B(Member) = 45k$$

  - Join cost: foreign-key and primary key join

$$B(User) + B(Member) = 100 + 5k = 5100$$

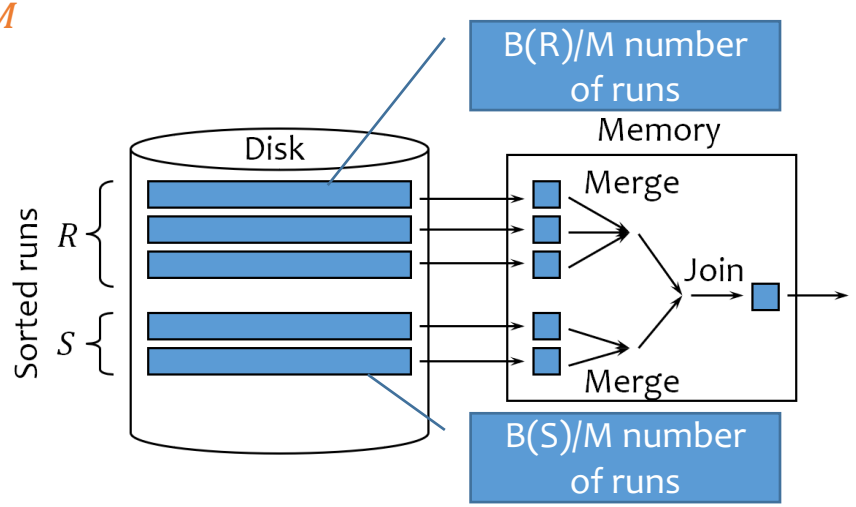
# Optimization of SMJ

- 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 phases:
  - I/O's: $3 \cdot \big(B(R) + B(S)\big)$
    - 1st phase: read B(R) + B(S) into memory for sorting and write sorted B(R) +B(S) to disk
    - 2nd phase: read B(R) + B(S) into memory to merge and join
  - Memory requirement
    - We must have enough memory to accommodate one block from each run: $M > \frac{B(R)}{M} + \frac{B(S)}{M}$
    - $M > \sqrt{B(R) + B(S)}$



# Performance of SMJ

- If SMJ completes in two passes:
  - I/O's: $3 \cdot \big(B(R) + B(S)\big)$
    - 1st phase: read B(R) + B(S) into memory for sorting and write sorted B(R) +B(S) to disk
    - 2nd phase: read B(R) + B(S) into memory to merge and join
  - 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(...)

# Outline

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

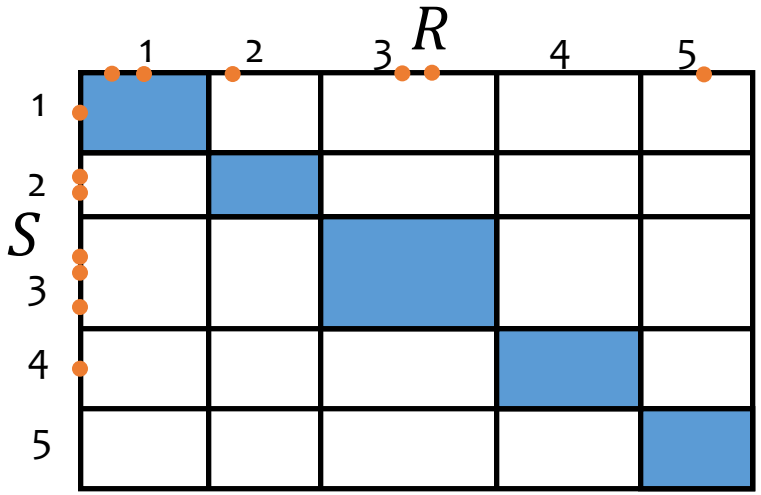
# Hashing-based algorithms


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

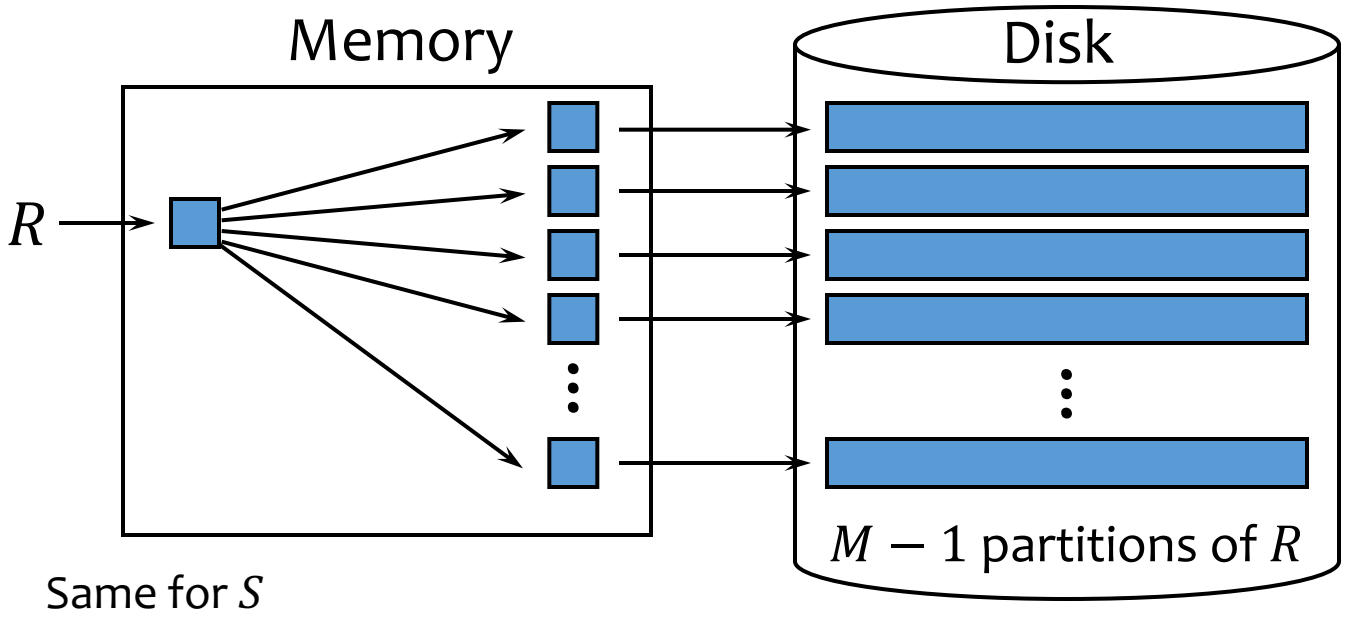



Nested-loop join considers all slots

Hash join considers only those along the diagonal!

# Partitioning phase

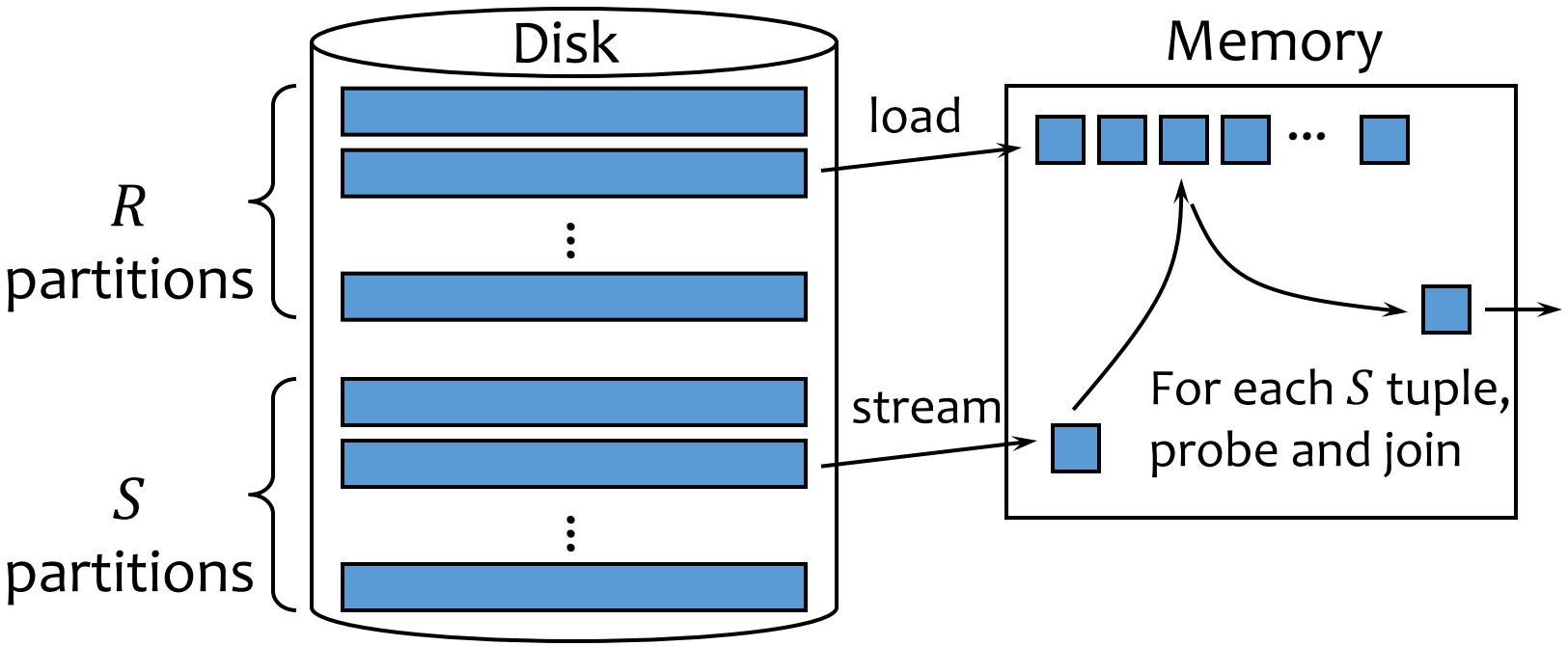
- Partition $R$ and $S$ according to the same hash function on their join attributes




Each partition has a size of B(R)/(M-1)

# 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!



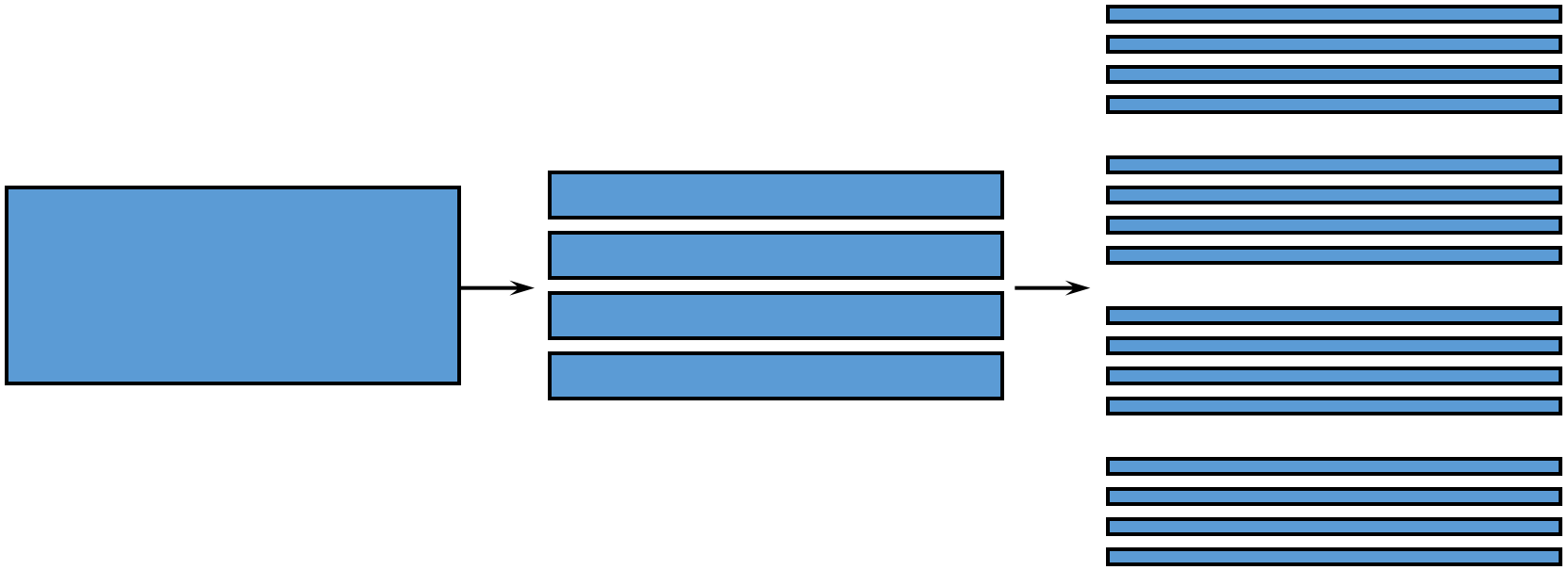
# Performance of (two-pass) hash join

- If hash join completes in two phases:
  - I/O's: $3 \cdot \big(B(R) + B(S)\big)$
    - 1st phase: read B(R) + B(S) into memory to partition and write partitioned B(R) +B(S) to disk
    - 2nd phase: read B(R) + B(S) into memory to merge and join
  - 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:

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

# Generalizing for larger inputs

- What if a partition is too large for memory?
  - Read it back in and partition it again!
  - Re-partition $O\left(\log_M B(R)\right)$ times



# 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

# Summary of techniques

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

# Another view of techniques

- Selection
  - Scan without index (linear search): $O(B(R))$
  - Scan with index – selection condition must be on search-key of index
    - B+ index: $O(\log(B(R))$
    - Hash index: $O(1)$
- Projection
  - Without duplicate elimination: $O(B(R))$
  - With duplicate elimination
    - Sorting-based: $O\left(B(R) \cdot \log_M B(R)\right)$
    - Hash-based: $O(B(R) + t)$where t is the result of the hashing phase
- Join
  - Block-based nested loop join (scan table): $O(B(R) \cdot \frac{B(S)}{M})$
  - Index nested loop join $O(B(R) + |R| \cdot (\text{index lookup}))$
  - Sort-merge join $O\left(B(R) \cdot \log_M B(R) + B(S) \cdot \log_M B(S)\right)$
  - Hash join $O\left(B(R) \cdot \log_M B(R) + B(S) \cdot \log_M B(S)\right)$