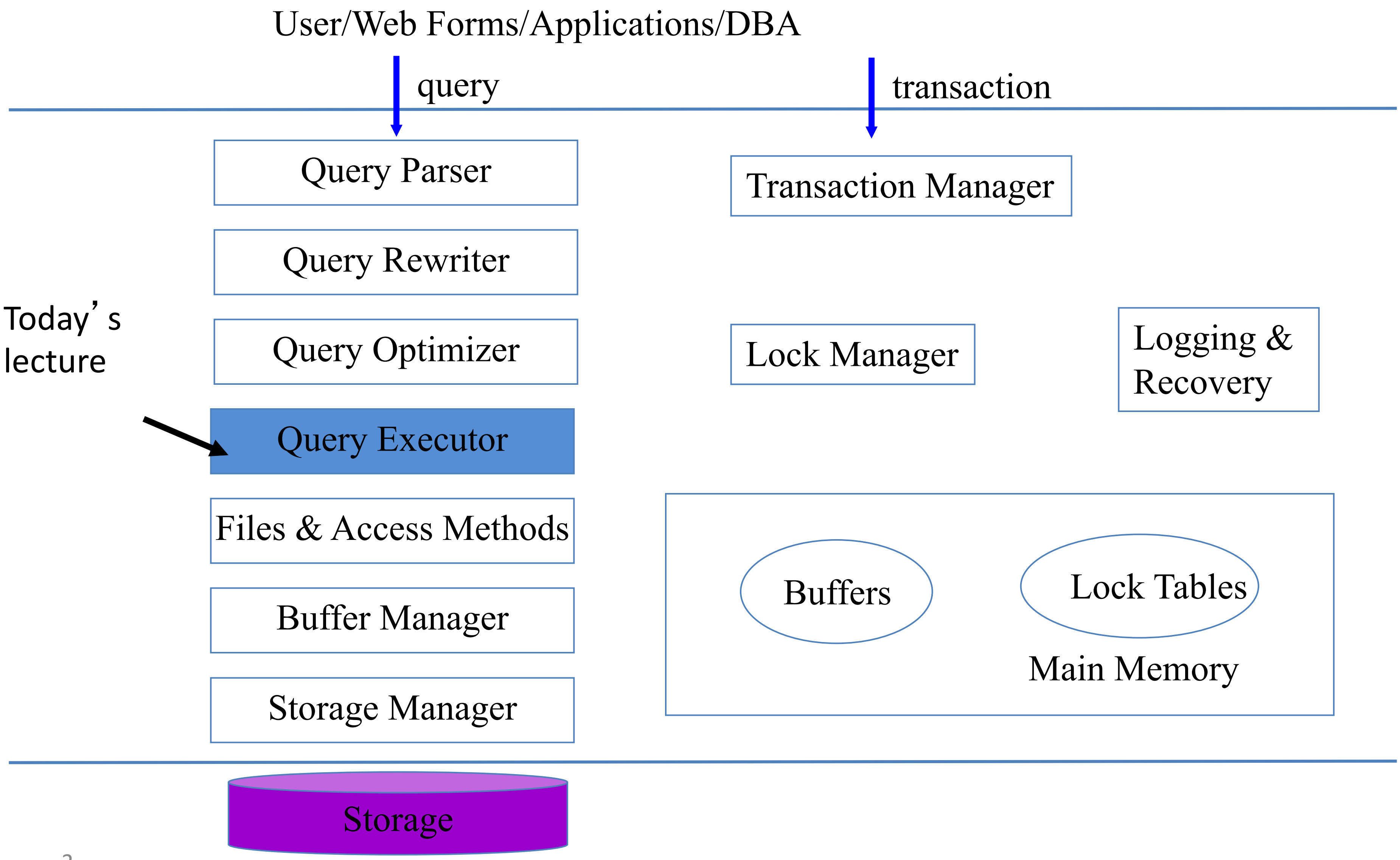
CS 540
Database Management Systems

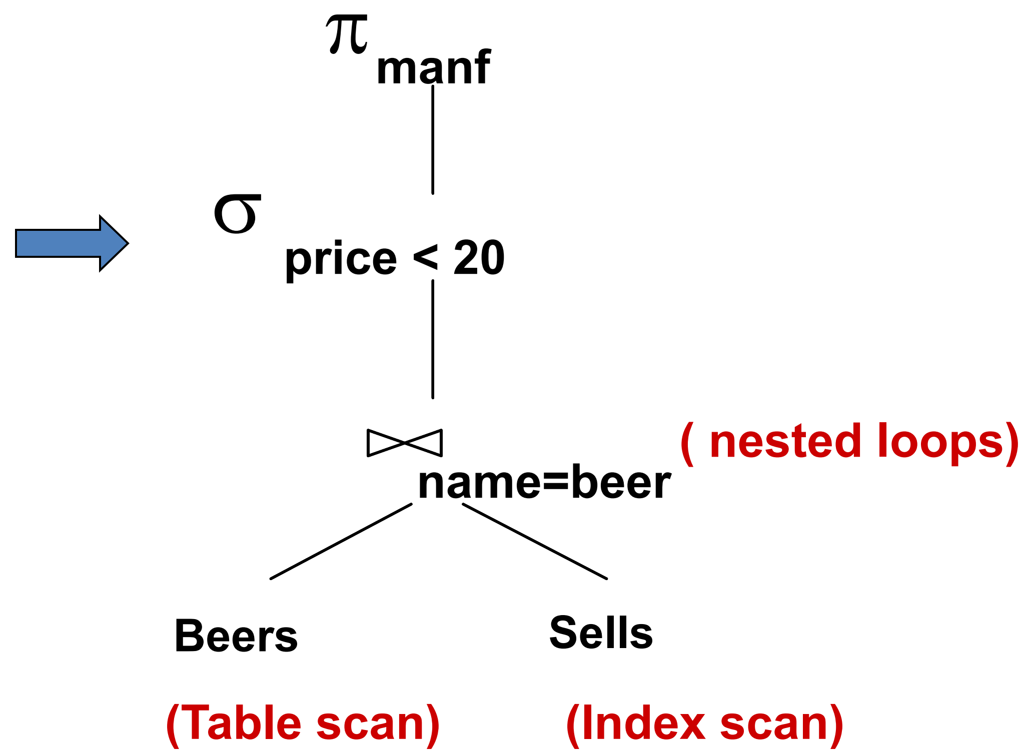
Query Processing

DBMS Architecture



Query Execution Plans

```
SELECT  B. manf
FROM    Beers B, Sells S
WHERE   B.name=S.beer AND
        S.price < 20
```



Query Plan:

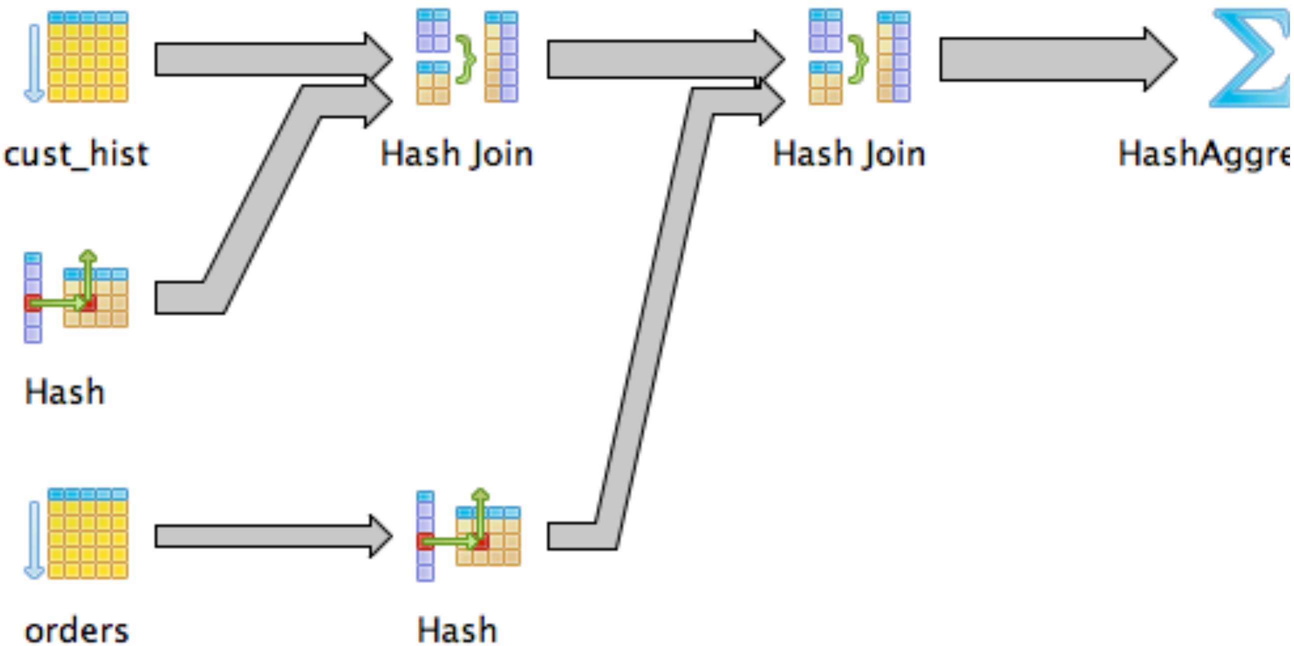
- logical plan (declarative)
- physical plan (procedural)
  - – procedural implementation of each logical operator
  - – scheduling of operations

Logical versus physical operators

- Logical operators
  - Relational Algebra Operators
    - Join, Selection, Projection, Union, …
- Physical operators
  - Algorithms to implement logical operators.
    - Hash join, nested loop join, …
- More than one physical operator for each logical operator

Explain command in Postgres

SELECT C.STATE, **SUM**(O.NETAMOUNT), **SUM**(O.TOTALAMOUNT)
FROM CUSTOMERS C
    JOIN CUST_HIST CH **ON** C.CUSTOMERID = CH.CUSTOMERID
    JOIN ORDERS O **ON** CH.ORDERID = O.ORDERID
GROUP BY C.STATE



Communication between operators: iterator model

- Each physical operator implements three functions:
  - **Open:** initializes the data structures.
  - **GetNext**: returns the next tuple in the result.
  - **Close**: ends the operation and frees the resources.
- It enables pipelining
- Other option: compute the result of the operator in full and store it in disk or memory:
  - inefficient.

Physical operators

- Logical operator: **selection**
  - read the entire or selected tuples of relation R.
    - tuples satisfy some predicate
- **Table-scan:** R resides in the secondary storage, read its blocks one by one.
- **Index-scan:** If there is an index on R, use the index to find the blocks.
  - more efficient
- Other operators for *join, union, group by*, ...
  - **join is the most important one.**
  - **focus of our lecture**

Both relations fit in main memory

- Internal memory join algorithms

- **Nested-loop join:** check for every record in *R* and every record in *S*; time = O(|*R*||*S*|)

- **Sort-merge join**: sort *R* and *S* followed by merging; time = O(|*S*|*log|*S*|) (if |R|<|S|)

- **Hash join**: build a hash table for *R*; for every record in *S*, probe the hash table; time =O(|*S*|) (if |R|<|S|)

External memory join algorithms

- At least one relation does not fit into main memory
- I/O access is the dominant cost
  - B(R): number of blocks of R.
  - |R| or T(R) : number of tuples in R.
- Memory requirement
  - M: number of blocks that fit in main memory

- **Example:** internal memory join algorithms : B(R) + B(S)
- We do not consider the cost of writing the output.
  - The results may be pipelined and never written to disk.

Nested-loop join of R and S

- For each block of R, and for each tuple r in the block:
  - For each block of S, and for each tuple s in the block:
    - Output rs if join condition evaluates to true over r and s

- R is called the outer table; S is called the inner table
- **cost**: $B(R) + |R| \cdot B(S)$
- **Memory requirement**: 4 (if *double buffering* is used)

- block-based nested-loop join
  - For each block of R, and for each block of S:

    For each r in the R block, and for each s in the S block: …

- **cost**: $B(R) + B(R) \cdot B(S)$
- **Memory requirement:** 4 (if *double buffering* is used)

Improving nested-loop join

- Use up the available memory buffers M
- Read M - 2 blocks from R
- Read blocks of S one by one and join its tuples with R tuples in main memory

- **Cost:** B(R) + [B(R) / (M – 2)] B(S)
  – almost B(R) B(S) / M
- **Memory requirement:** M

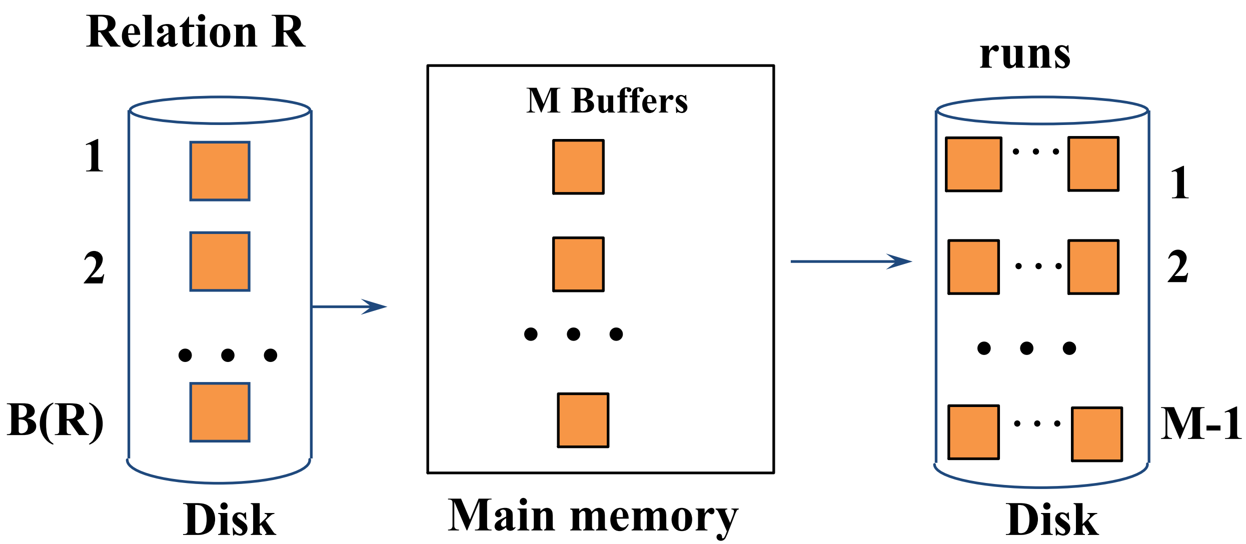
Index-based (zig-zag) join

- Join R and S on R.A = S.B
- Use ordered indexes over R.A and S.B to join the relations.
  - B+ tree
  - Use current indexes or build new ones.
  - Cost: B(R) + B(S)
- Memory requirement?

Index-based join algorithm

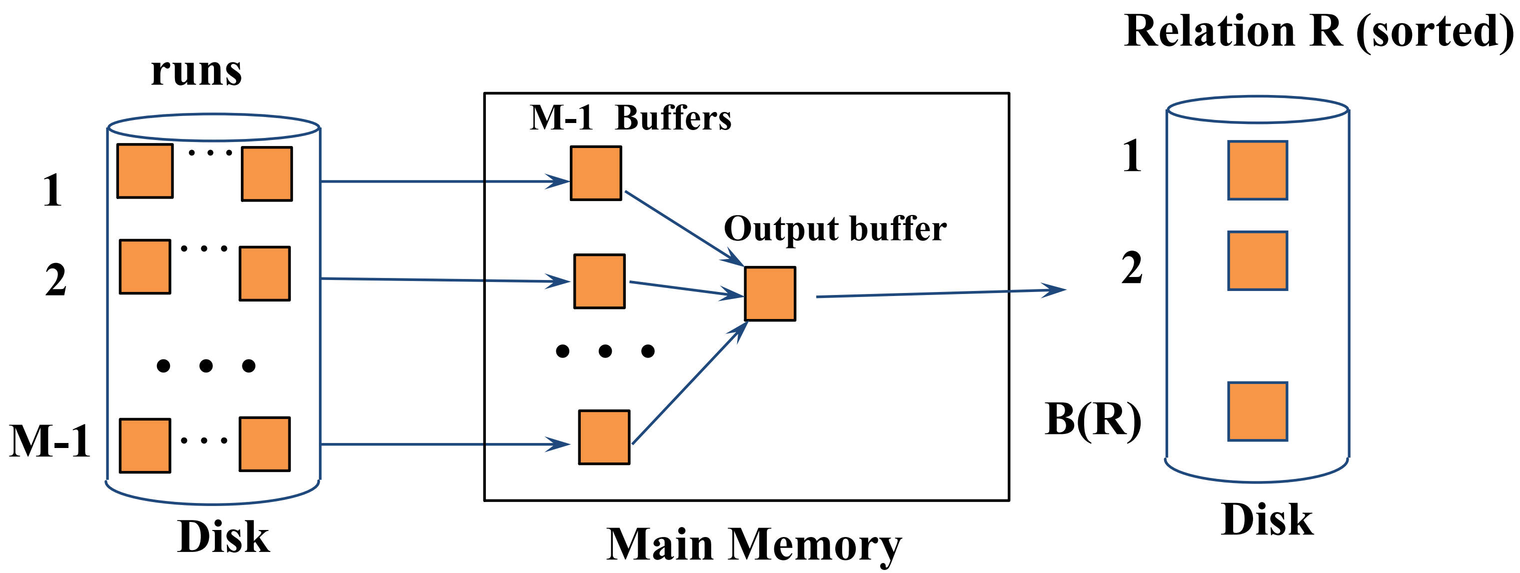
- Only R has an index over the join attribute.
- Read S, for each tuple of S find matching tuples in R.
- If S does not share its blocks with other relations
  - V(R,A): Number of distinct values of attribute A in R.
  - Clustered index on R: B(S) + T(S) B(R) / V(R,A).
  - Unclustered index on R: B(S) + T(S) T(R) / V(R,A).
- Efficiency
  - If S is small or V(R,A) is very large, not need to examine all tuples in R.
    - more efficient than nested-loop.

Two pass, multi-way merge sort



- **Problem:** sort relation R that does not fit in main memory
- Phase 1: Read R in groups of M blocks, sort, and write them as runs of size M on disk.

Two pass, multi-way merge sort



- Phase 2: Merge M – 1 blocks at a time and write the results to disk.
  - Read one block from each run.
  - Keep one block for the output.

Two pass, multi-way merge Sort

- **Cost:** 2B(R) in the first pass + B(R) in the second pass.

- **Memory requirement**: M
  - B(R) <= M (M – 1) or simply B(R) <= M^2

General multi-way merge sort

- Input: 1, 7, 4, 5, 2, 8, 9, 6, 3, 0
- Each block holds one number, and memory has 3 blocks
- Pass 0
  - 1, 7, 4 ->1, 4, 7
  - 5, 2, 8 -> 2, 5, 8
  - 9, 6, 3 -> 3, 6, 9
  - 0 -> 0
- Pass 1
  - 1, 4, 7 + 2, 5, 8 -> 1, 2, 4, 5, 7, 8
  - 3, 6, 9 + 0 -> 0, 3, 6, 9
- Pass 2 (final)
  - 1, 2, 4, 5, 7, 8 + 0, 3, 6, 9 -> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

General multi-way merge sort

- Pass 0: read M blocks of R at a time, sort them, and write out a level-0 run
  - There are $\lceil B(R) / M \rceil$ 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 = # of level-$(i$–1) runs / $(M - 1)$
- Final pass produces 1 sorted run

Analysis of general multi-way merge sort

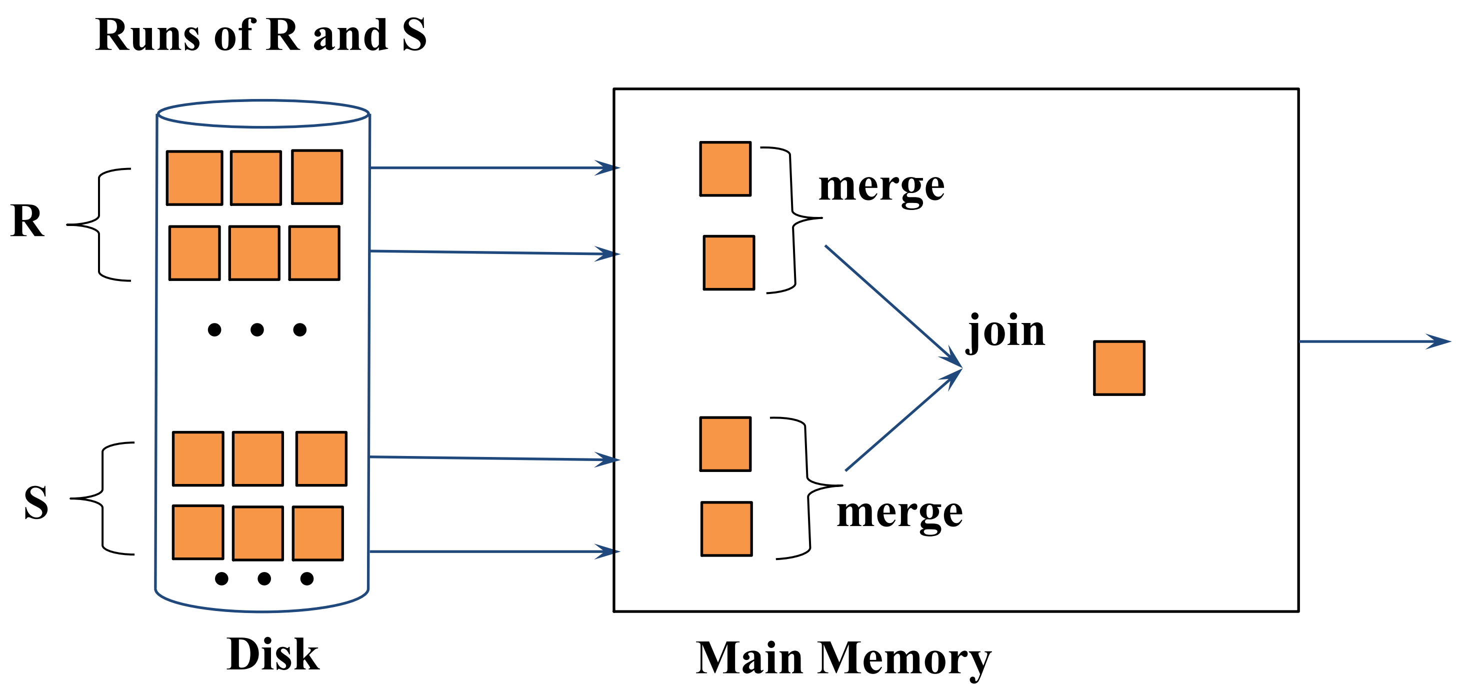
- Number of passes: $\left\lceil \log_{M-1} \left\lceil B(R)/M \right\rceil \right\rceil + 1$
- **cost**
  - #passes · 2 · $B(R)$: each pass reads the entire relation once and writes it once
  - Subtract $B(R)$ for the final pass
  - Simply $O(\ B(R) \cdot \log_M B(R)\ )$
- **Memory requirement**: M

Sort-merge join algorithm

- Sort R and S according to the join attribute, then merge them
  - sort R and S using two pass multi-way merge sort
  - *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*
- **Cost**: sorting + 2 B(R)+ 2 B(S)
- What if more than M blocks match on join attribute?
  - use nested loop join algorithm
  - B(R) B(S) if everything joins
- **Memory Requirement**: $B(R) <= M^2$, $B(S) <= M^2$

Optimized sort-merge join algorithm

- Combine join with the merge phase of sort
  - Sort R and S in M runs (overall) of size M on disk.
  - Merge and join the tuples in one pass.



Optimized sort-merge join algorithm

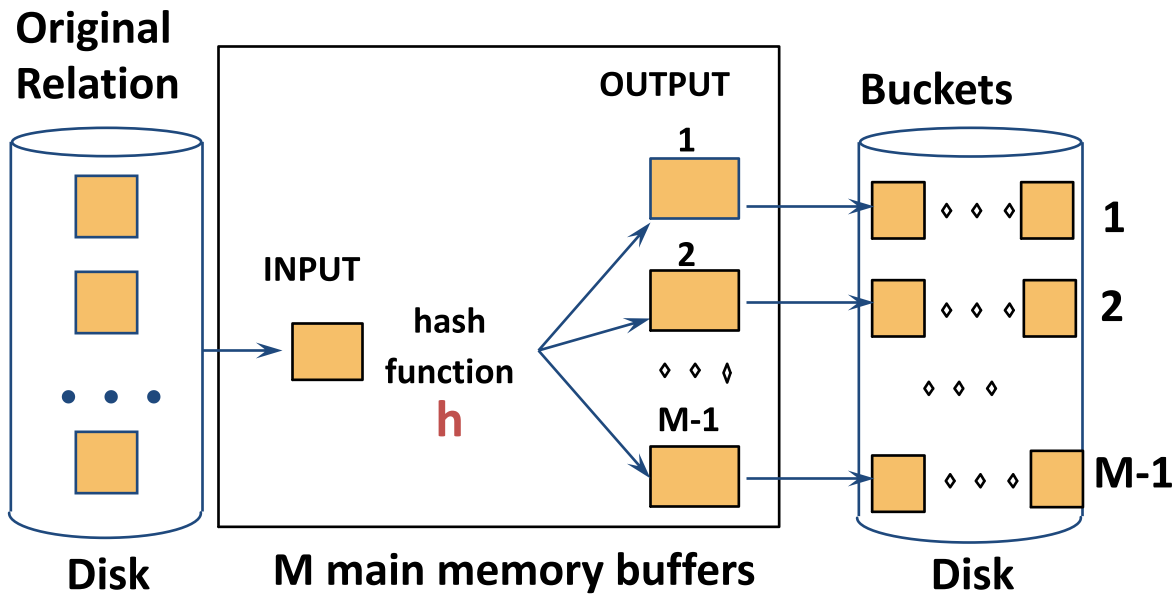
- **Cost**: $3B(R) + 3B(S)$
- **Memory Requirement**: $B(R) + B(S) <= M^2$
  - because we merge them in one pass
- More efficient but more strict requirement.

(Partitioned) Hash join or R and S

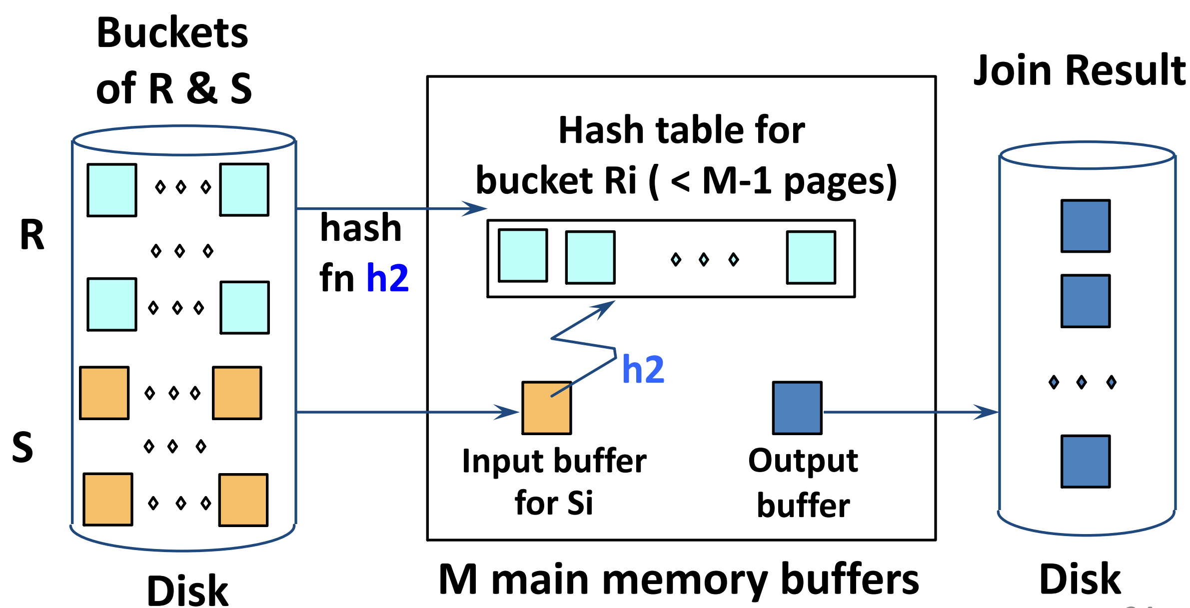
- Step 1:
  - Hash S into M buckets
  - send all buckets to disk
- Step 2
  - Hash R into M buckets
  - Send all buckets to disk
- Step 3
  - Join corresponding buckets
    - If tuples of R and S are not assigned to corresponding buckets, they do not join

Hash Join

- Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.

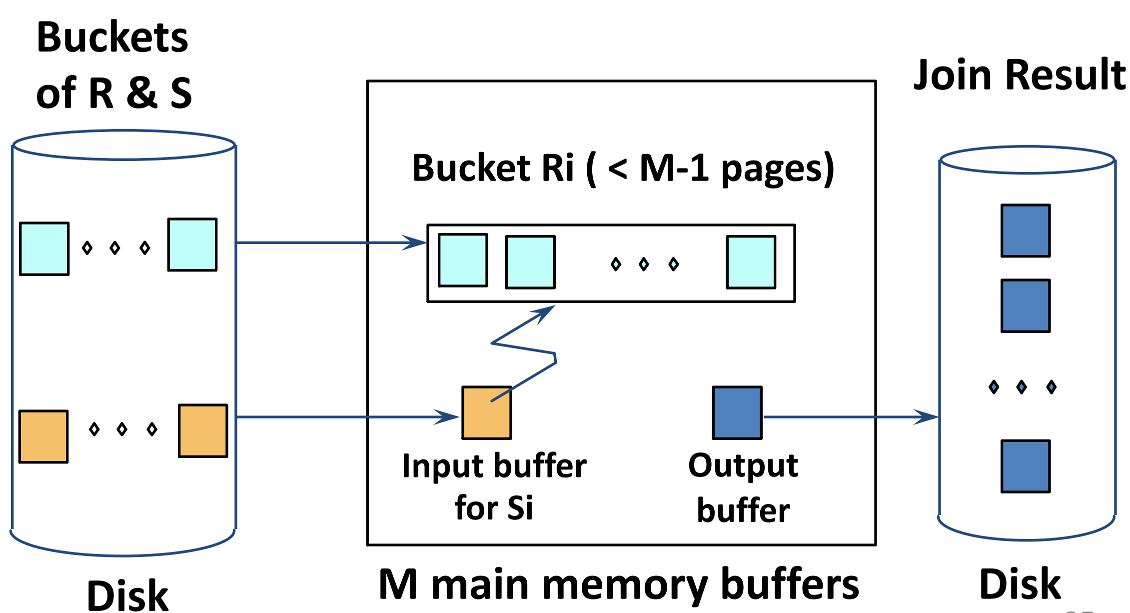


- Read in a partition of R, hash it using h2 (<> h!). Scan matching partition of S, search for matches.



Hash join

- **Cost**: 3 B(R) + 3 B(S).
- **Memory Requirement**:
  - The smaller bucket must fit in main memory.
  - Let min(B(R), B(S)) = B(R)
  - B(R) / (M – 1) <= M, roughly B(R) <= M^2



Handle partition overflow

- **Overflow on disk**: an *R* partition is larger than memory size
  – Solution: recursive partition.

Hash-based versus sort-based join

- **Hash join:** smaller amount of main memory
  - $\text{sqrt}(\min(B(R), B(S))) < \text{sqrt}(B(R) + B(S))$
  - Hash join wins if the relations have different sizes
- Hash join performance depends on the quality of hashing
  - Hard to generate balanced buckets
- Sort-based join wins if the relations are in sorted order
- Sort-based join generates sorted results
  - useful when there is **Order By** in the query
  - useful the following operators need sorted input
- Sort-based join can handle inequality join predicates

Duality of Sort and Hash

- Divide-and-conquer paradigm
- Handling very large inputs
  - Sorting: multi-level merge
  - Hashing: recursive partitioning