Joins on a single machine

csc343, Introduction to Databases
Renée Miller and Diane Horton
Fall 2016

Slides adapted from Ramakrishnan and Gerhke
pages.cs.wisc.edu/~dbbook/

UNIVERSITY OF TORONTO

Road Map

- Basics of how to execute SQL queries on a single machine
  - Introduction to SQL Join processing
- Basics of how to scale SQL to multiple machines
  - Introduction to Parallel SQL Join processing
- Basics of how to scale simple analysis to massive numbers of machines
  - Introduction to Map-Reduce

For more information

- Database Management Systems. Ramakrishnan and Gehrke. Third Edition. Chapter 22.1.

- Mining of Massive Datasets. Jure Leskovec, Anand Rajaraman, and Jeff Ullman.
  - http://infolab.stanford.edu/~ullman/mmds/ch2.pdf

Three phases of query execution

- 1. Parsing: Produces a parse tree.
- 2. Query rewrite:
  - From the parse tree, constructs an abstract query execution plan, called a logical query plan.
  - This is usually an algebraic representation of the query.
  - Then rewrites into an equivalent but more efficient plan.
- 3. Physical plan generation:
  - Chooses order of execution of each operator.
  - Chooses an algorithm for each operator.
  - Represented as an expression tree.

Query optimization

- Phases 2 and 3 are called query optimization.
- Many choices must be made.
- Choices are informed by metadata, such as:
  - size of each table
  - distribution of values in each table
  - existence of indexes
  - layout of data on disk
- Let's examine this for one operator: Join.

Aside: Indexes

- One way to get faster access to data: keep it in a balanced search tree.
- If you want fast access by more than one attribute?
  - Keep all data at the leaves.
  - Values in internal nodes are just guides to the proper leaf.
  - Have more than one tree pointing to all those leaves.
  - Each tree is organized by different attribute(s).
- We call such a tree an index on the data.

Must understand storage first

- To assess algorithm speed, we must understand costs.
- In previous courses
  - Atomic operations like arithmetic operations, assignment, and following a pointer were O(1).
  - We thought about how many times they occurred.
- Now our data doesn't fit in memory.
  - Some operations will require disk input/output.
  - It is orders of magnitudes slower.
- We need to know more about this.

Implications of using disk storage

- DBMS stores information on (“hard”) disks.
  - Greater capacity than main memory (RAM)
  - Higher latency: delay from request to desired outcome.
- This has major implications for DBMS design!
  - READ: transfer data from disk to RAM.
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Data transfer is in large chunks

- Overhead to get ready to transfer data is much greater than transfer time.
- So once we incur the overhead, might as well retrieve a big chunk.
  - Data is stored and retrieved in units called disk blocks or pages.
- This only provides a benefit if we later will need the other data that was retrieved.
- So the DBMS organizes the data in page-sized chunks.
  - Data structures become “file structures”!

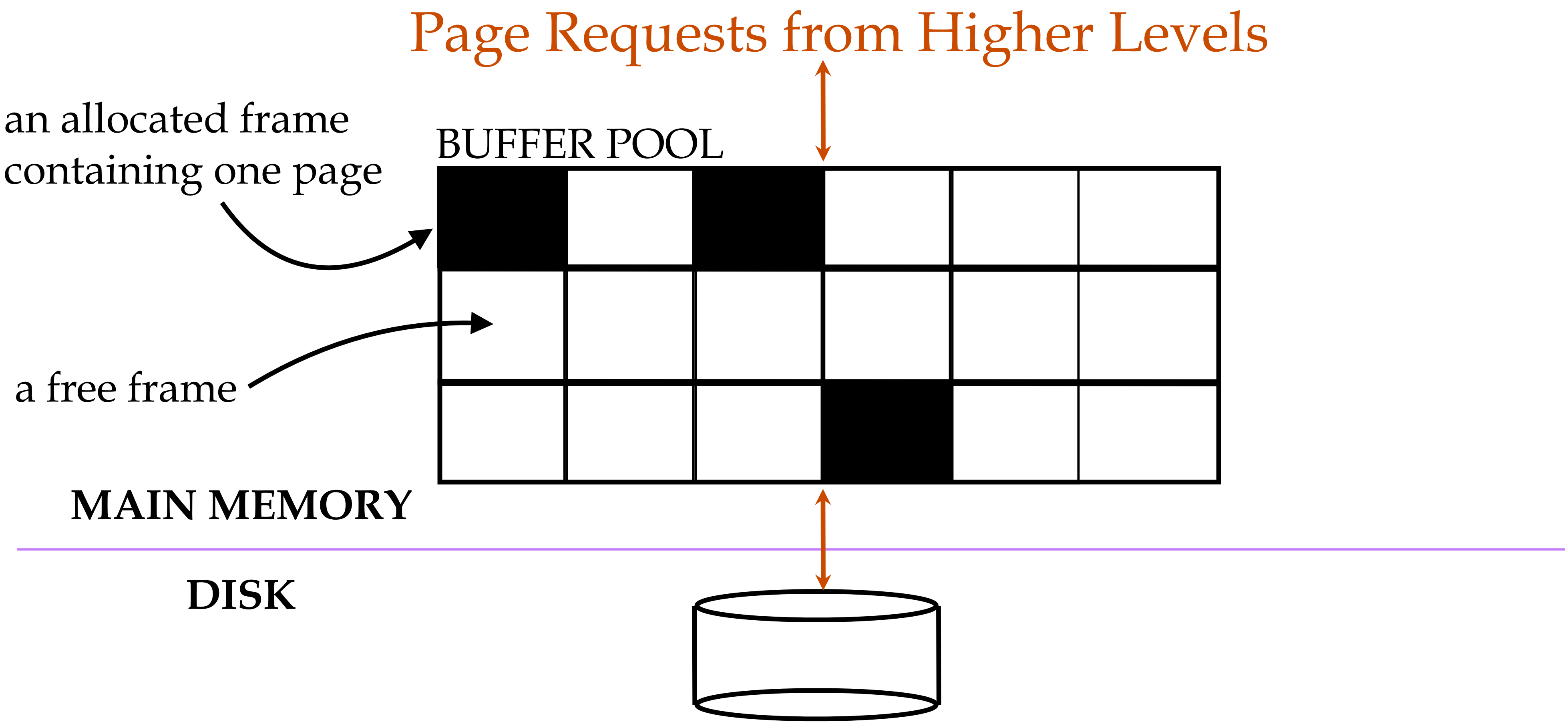
Placement of pages matters

- Time to retrieve a disk page varies depending upon its location on disk.
  - (This is not so for RAM.)
- Therefore, relative placement of pages on disk has major impact on DBMS performance!

Buffering

- The strategy of bringing a whole page into memory and working within it as much as possible is called buffering.
- A DBMS maintains not one but a set of buffers in memory, called a buffer pool.
- When a new page must be read from disk
  - It goes in a free frame in the buffer pool, if one exists.
  - If not, an existing buffer must be overwritten. The replacement policy decides which.
  - The overwritten buffer must first be rewritten to disk if it has changed.

Buffer Management in a DBMS



A table of <frame#, pageid> pairs is maintained.

Layered architecture

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Higher levels don't need to know how this is done, or how free space is managed.

Doesn't the OS do buffering?

- An operating system uses buffering also.
  - Allows it to provide the illusion that you have much more RAM than you do.
- But a DBMS can do a better job managing blocks on disk and in the buffer pool.
  - It has meta-data that lets it pick the best strategies given the state of the database.
- So the DBMS takes control of these matters from the OS.

Back to implementing Join

- We are now ready to consider and compare several algorithms for Join.

Schema for Examples

- Domain: a sailing club in which members can reserve sailboats.

Sailors (<u>**sid**: integer</u>, **sname**: string, **rating**: integer, **age**: real)

Reserves (<u>**sid**: integer, **bid**: integer, **day**: dates</u>, **rname**: string)

Reserves[sid] $\subseteq$ Sailors[sid]
Reserves[bid] $\subseteq$ Boats[bid] *but we won't use the Boats relation*

“Equality Joins” are common


```
SELECT *
FROM Reserves R, Sailors S
WHERE R.sid = S.sid
```

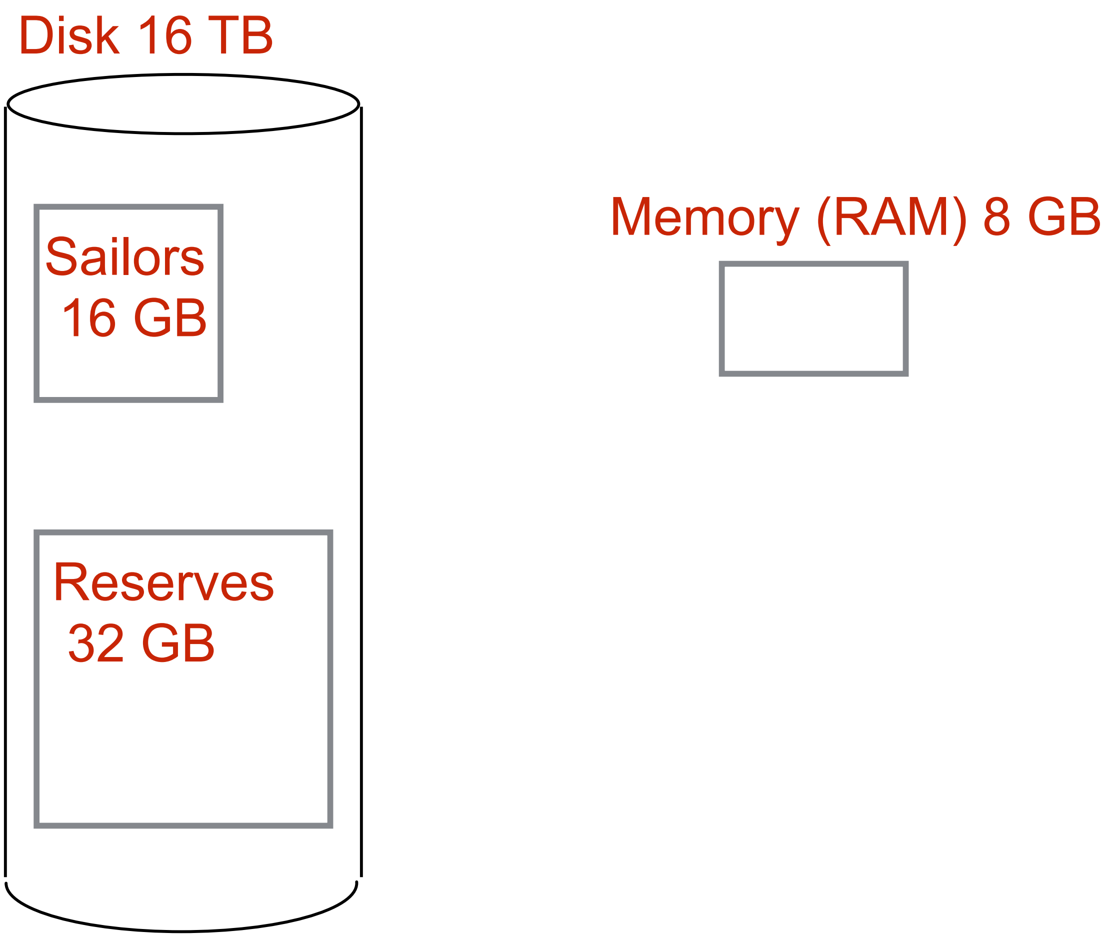
- In relational algebra, this query is R ⋈ S.
- Natural join and other equality joins are common!
  - So must be carefully optimized.
- Straightforward implementation:
  - R X S
  - equality selection R.sid = S.sid
- But R X S is large; so this is inefficient.

Tuple-based Nested-Loop Join


```
for each tuple r in R:
    for each tuple s in S:
        if r and s match on the relevant attributes:
            add <r, s> to result
```

- This could require $T_R \times T_S$ disk I/Os!
- But we can't assume both relations fit in memory.

Data too large for memory?



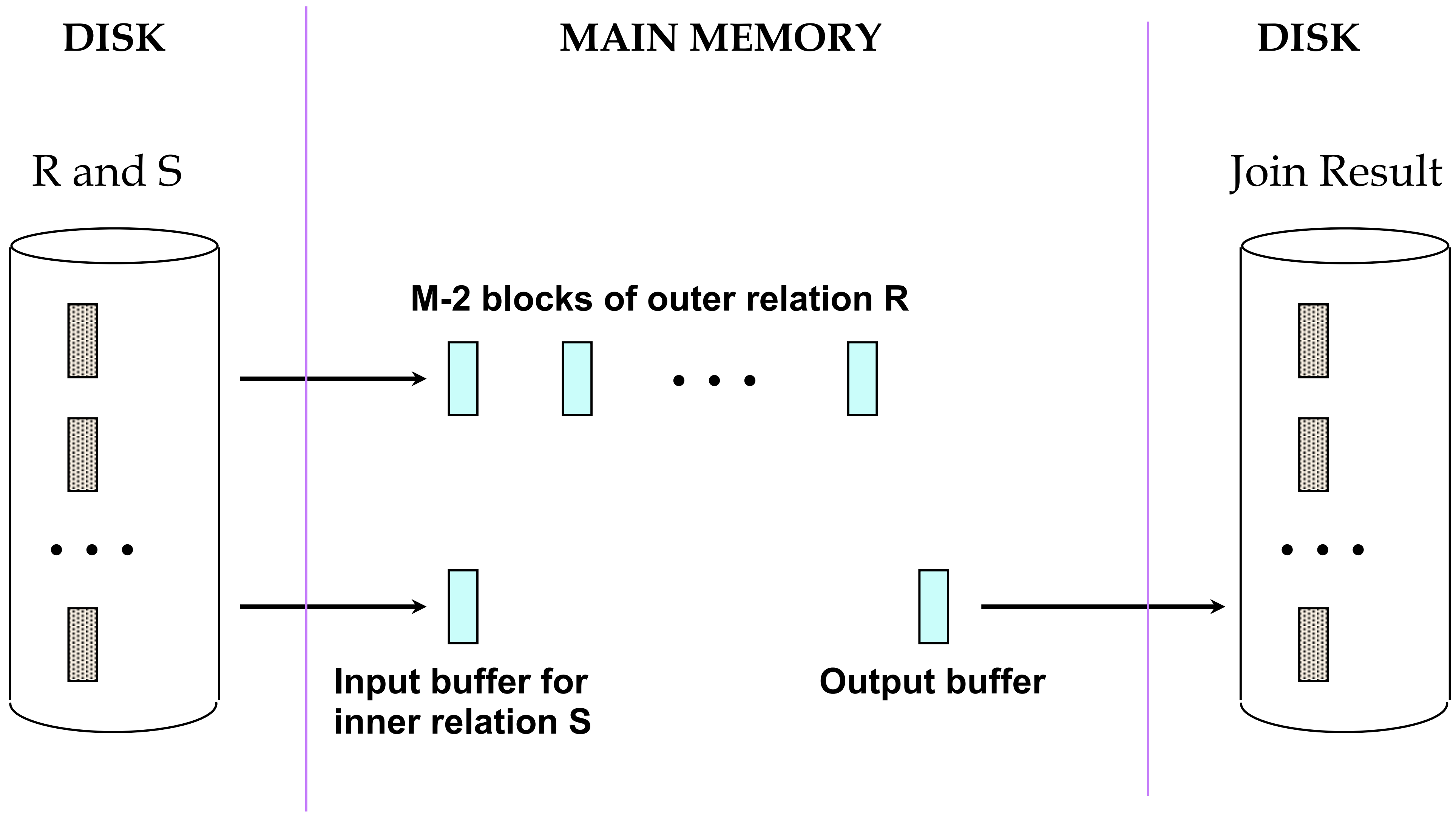
So how to implement join?

- We saw that
  - Data may well not fit in memory.
  - Realities of disk storage and buffering must be taken into account if we want speed.
- So how should we implement the join operator?

Block-Based Nested-Loop Join

- We need to use the buffers to hold portions of the tables.
- Storage strategy:
  - Use one page as an input buffer for scanning the outer R.
  - Use one page as the output buffer.
  - Use all M-2 remaining pages to hold chunks (of multiple blocks) of inner S.
- Algorithm:
  - Load as many blocks of S as possible into those M-2 buffers.
  - For each block of R, load it into a buffer and output all possible joined tuples from that much of R and S.
  - Repeat for the remaining chunks of S.

Block-Based Nested-Loop Join



Block-Based Nested-Loop Join


```
for each chunk of M-2 blocks of R:
    read these blocks into buffers in main memory
    for each block b of S:
        read b into a buffer in main memory
        for each tuple t1 of b:
            for each tuple t2 of R in memory:
                if t2 joins with t1:
                    add the join of t1 and t2 to the result
```

Block-Based Nested-Loop Join


```
for each chunk of M-2 blocks of R:
    read these blocks into buffers in main memory
    for each block b of S:
        read b into a buffer in main memory
        for each tuple t1 of b:
            for each tuple t2 of R in memory:
                if t2 joins with t1:
                    add the join of t1 and t2 to the result
```

For each block of S, load it into a buffer and output all possible joined tuples from that much of R and S.

Block-Based Nested-Loop Join


```
for each chunk of M-2 blocks of R:
    read these blocks into buffers in main memory
    for each block b of S:
        read b into a buffer in main memory
        for each tuple t1 of b:
            for each tuple t2 of R in memory:
                if t2 joins with t1:
                    add the join of t1 and t2 to the result
```

We have a tuple from each relation. If they join, they go in the result.

How many disk I/Os?

- Suppose
  - R has 1,000 blocks
  - S has 500 blocks
  - We have 102 buffer frames.
- How many disk I/Os are needed to compute R ⋈ S?

If relation R is the outer relation

- I.e., R is the relation in the outer loop.
- We have 100 buffer frames to hold 100-block chunks of R.
  - So R will be broken into 10 of these 100-block chunks.
  - (The outer loop will iterate 10 times.)
  - Across these iterations, we will do 1,000 IOs for reading R.
- For each of the 10 iterations, we read every block of S.
  - S has 500 blocks.
  - Across these iterations, we will do 10 x 500 IOs for reading S.
- Total IOs = 1,000 + 5,000 = 6,000

R has 1,000 blocks

S has 500 blocks

102 buffer frames

If relation S is the outer relation

- I.e., R is the relation in the outer loop.
- We have 100 buffer frames to hold 100-block chunks of S.
  - So S will be broken into 5 of these 100-block chunks.
  - (The outer loop will iterate 5 times.)
  - Across these iterations, we will do 500 IOs for reading S.
- For each of the 5 iterations, we read every block of R.
  - R has 1,000 blocks.
  - Across these iterations, we will do 5 x 1,000 IOs for reading R.
- Total IOs = 500 + 5,000 = 5,500

R has 1,000 blocks

S has 500 blocks

102 buffer frames

General cost of Sort-Merge Join

- Suppose R has B(R) blocks and S has B(S) blocks.
- The outer loop iterates $\lceil B(R) / (M-2) \rceil$ times.
- On each iteration, it must read
  (M-2) blocks of R and B(S) blocks of S
- So in total, we read approximately this many blocks:
  [B(R) x (M - 2 + B(S))] / (M-2)
- Assuming M << B(R) and M << B(S), this is approximately:
  [B(R) x B(S)] / M disk I/Os
- It turns out we can do better.

Sort-Merge Join

- Strategy:
  - Sort R and Sort S.
  - Use a merge algorithm to find all tuples that join.
  - During the merge, when finding all tuples of R and S that share a certain value for the join attribute (or attributes),
    - Bring into buffer frames every block of R and every block of S that includes that value.
- Benefit:
  - During the merge, you never have to load any block more than once.
- This assumes M-1 buffer frames are sufficient for merging.

Cost of Sort-Merge Join

- Sorting a relation T can be done with only 4 x B(T) disk I/Os.
- So to sort R and S requires 4 x [B(R) + B(S)] disk I/Os.
- Because merging can be done without loading any block more than once, it takes B(R) + B(S) disk I/Os.
- Grand total: 5 x [B(R) + B(S)] disk I/Os.
- In our concrete example, that is 6,000 disk I/Os.

- Much better than Block-Based Nested-Loop Join.
- But we can do even better!

R has 1,000 blocks

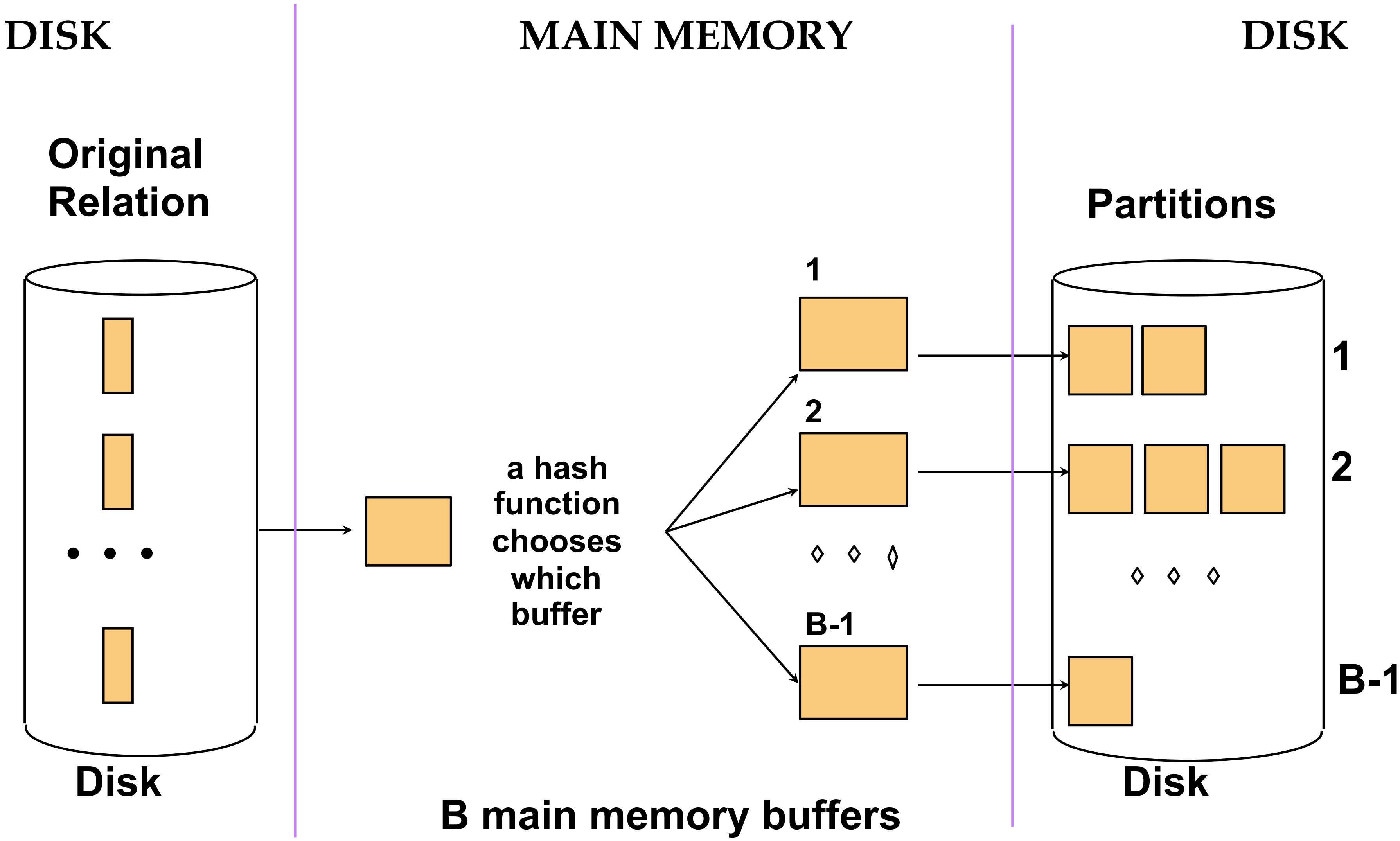
S has 500 blocks

101 buffer frames

Hash Join

- Phase 1
  - Use a hash function to
    - distribute the tuples of R across B-1 partitions
    - distribute the tuples of S across B-1 partitions
  - The hash function is based on the join attribute(s)
  - It guarantees that tuples r from R and s from S will join iff they are in the same partition.
- Phase 2
  - For each partition, use a merge algorithm to find all tuples that join.

Partitioning phase



Cost to partition a relation

- for each block of relation T:
  - read the block
  - for each tuple t in the block:
    - hash t to choose a partition
    - if the buffer for that partition is full, write it to disk
    - write t to the buffer for that partition
  - write any non-empty buffers to disk
- Requires one read and one write per block.
- So cost is 2 x B(T) disk I/Os.

Cost for Hash Join

- Cost to partition R is 2 x B(R) I/Os.
- Cost to partition S is 2 x B(S) I/Os.
- Merging takes B(R) + B(S) I/Os as before.
- Grand total = 3 x [B(R) + B(S)]

- In our concrete example, that is 4,500 disk I/Os.

R has 1,000 blocks

S has 500 blocks

101 buffer frames

Comparison of Join Implementations

	In our Example	In General
Block-Based Nested-Loop Join	4,500 or 6,000	[B(R) x B(S)] / M
Sort-Merge Join	6,000	5 x [B(R) + B(S)]
Hash Join	4,500	3 x [B(R) + B(S)]

R has 1,000 blocks

S has 500 blocks

101 buffer frames

Comparison of Join Implementations

- Block-Based Nested-Loop Join is generally slowest.
  - It just *happened* to come out relatively well in our example.
- Hash Join is generally fastest.
- But if you already have indices that let you access both R and S in sorted order according to the join attributes,
  - Sort-Merge Join takes advantage of this.
  - Total time is just 1 x [B(R) + B(S)] — far better!