UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems

Lecture 2: Join

Xiangyao Yu
9/13/2021

Today's Paper: Join

Join Processing in Database Systems with Large Main Memories

LEONARD D. SHAPIRO
North Dakota State University

We study algorithms for computing the equijoin of two relations in a system with a standard architecture but with large amounts of main memory. Our algorithms are especially efficient when the main memory available is a significant fraction of the size of one of the relations to be joined; but they can be applied whenever there is memory equal to approximately the square root of the size of one relation. We present a new algorithm which is a hybrid of two hash-based algorithms and which dominates the other algorithms we present, including sort-merge. Even in a virtual memory environment, the hybrid algorithm dominates all the others we study.

Finally, we describe how three popular tools to increase the efficiency of joins, namely filters, Babb arrays, and semijoins, can be grafted onto any of our algorithms.

Categories and Subject Descriptors: H.2.0 [**Database Management**]: General; H.2.4 [**Database Management**]: Systems—*query processing*; H.2.6 [**Database Management**]: Database Machines

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Hash join, join processing, large main memory, sort-merge join

ACM Transactions on Database Systems, 1986

Agenda

System architecture and assumptions

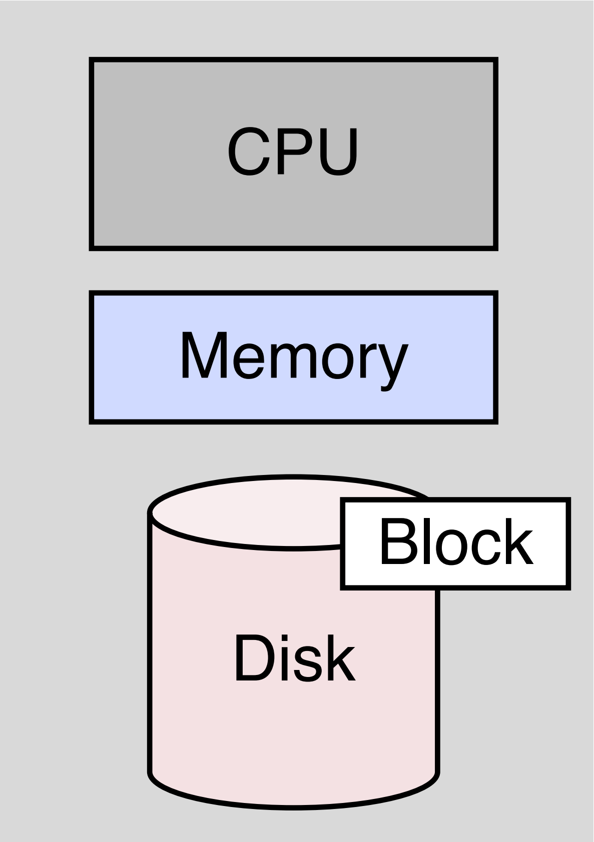
Notations

Join algorithms

- Sort merge join
- Simple hash join
- GRACE hash join
- Hybrid hash join

Partition overflow and additional techniques

System Architecture and Assumptions



CPU: uniprocessor

- Avoids sync complexity
- Could be built on systems of the day

Memory

- Tens of Megabytes

Focus only on equi-join

Notation

Relations: R, S ($|R| < |S|$)

Join: $S \bowtie R$

Memory: M

$|R|$: number of blocks in relation R (similar for S and M)

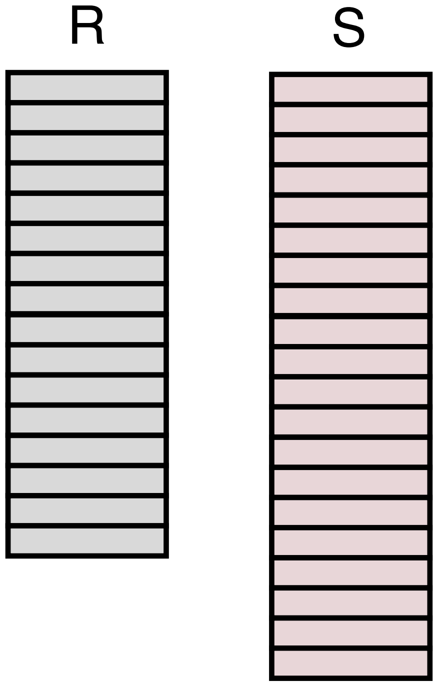
F: hash table for R occupies $|R| * F$ blocks

Join Algorithms

Sort Merge Join

Phase 1: Produce sorted runs of S and R

Phase 2: Merge runs of S and R, output join result

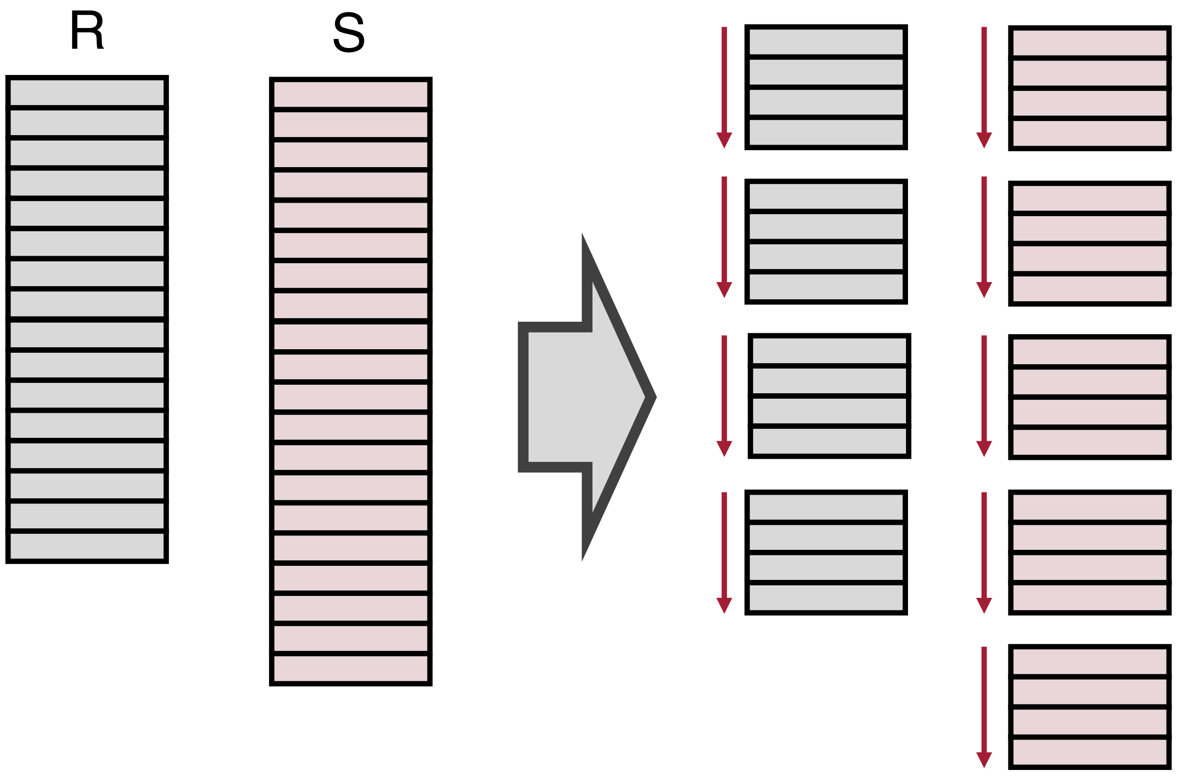


Unsorted R and S

Sort Merge Join

Phase 1: Produce sorted runs of S and R

Phase 2: Merge runs of S and R, output join result

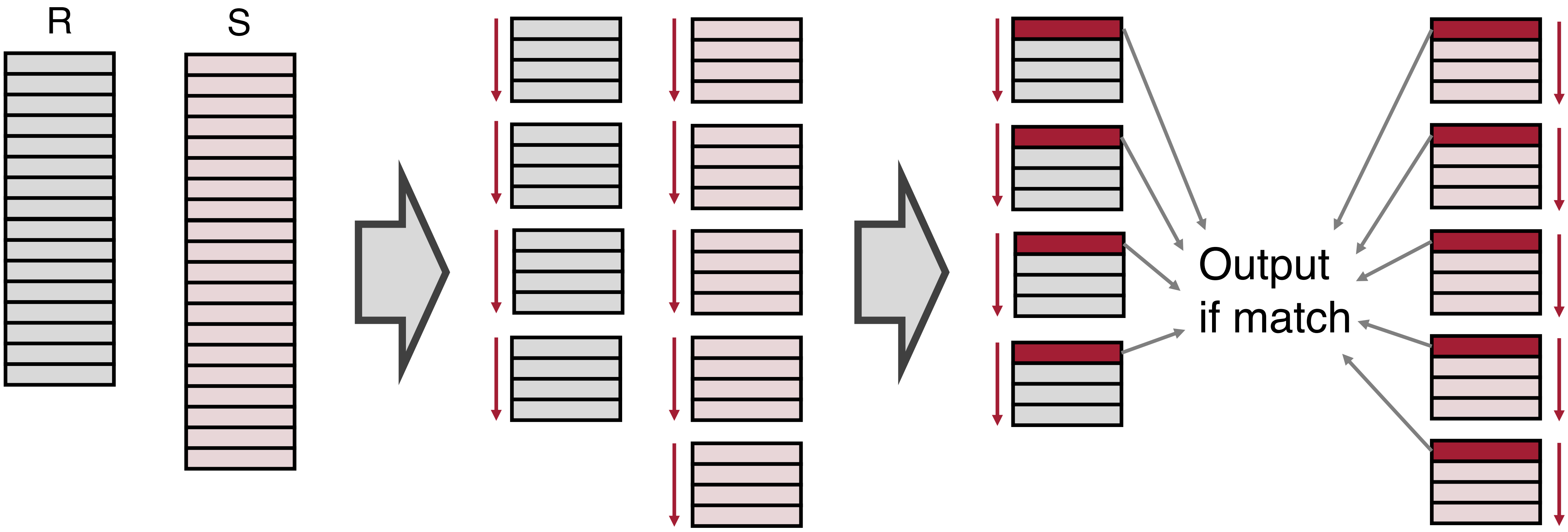


Unsorted R and S   Sorted runs of R and S

Sort Merge Join

Phase 1: Produce sorted runs of S and R

Phase 2: Merge runs of S and R, output join result



Unsorted R and S

Sorted runs of R and S

Find matches in sorted runs

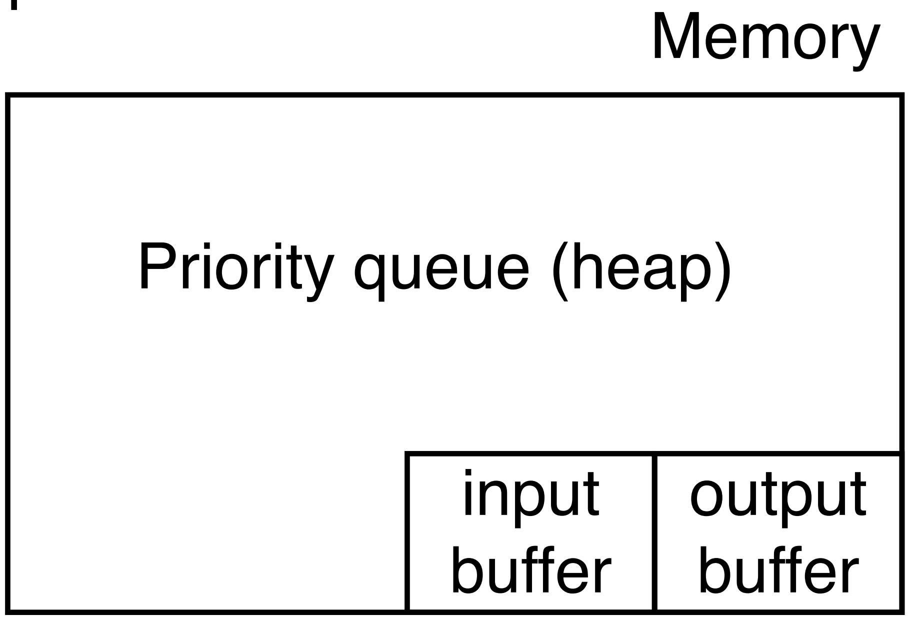
Sort Merge Join – Phase 1

Phase 1: Produce sorted runs of S and R

- Each run of S will be **2 × | M |** average length

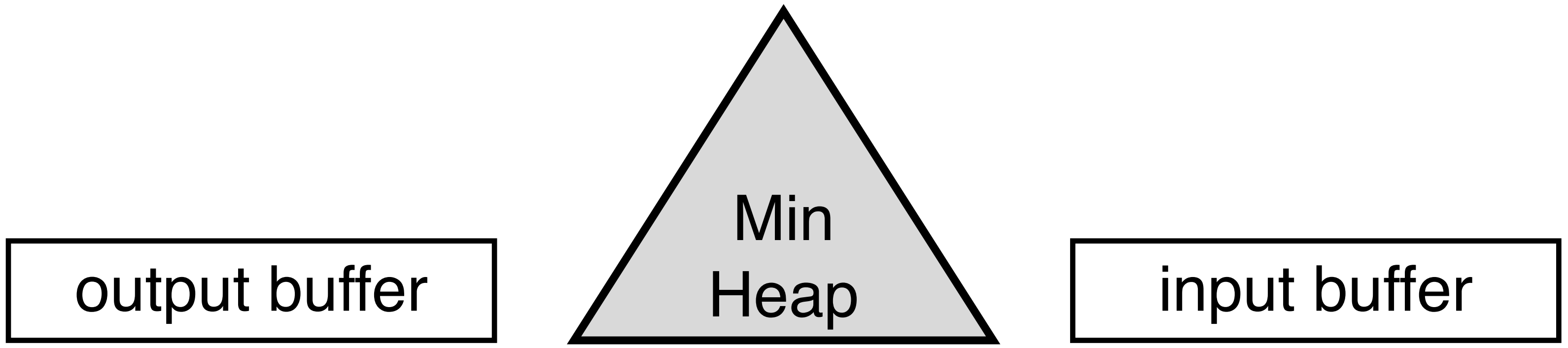
Q: Where does 2 come from?

A: Replacement selection



Memory layout in Phase 1

Sort Merge Join – Replacement Selection

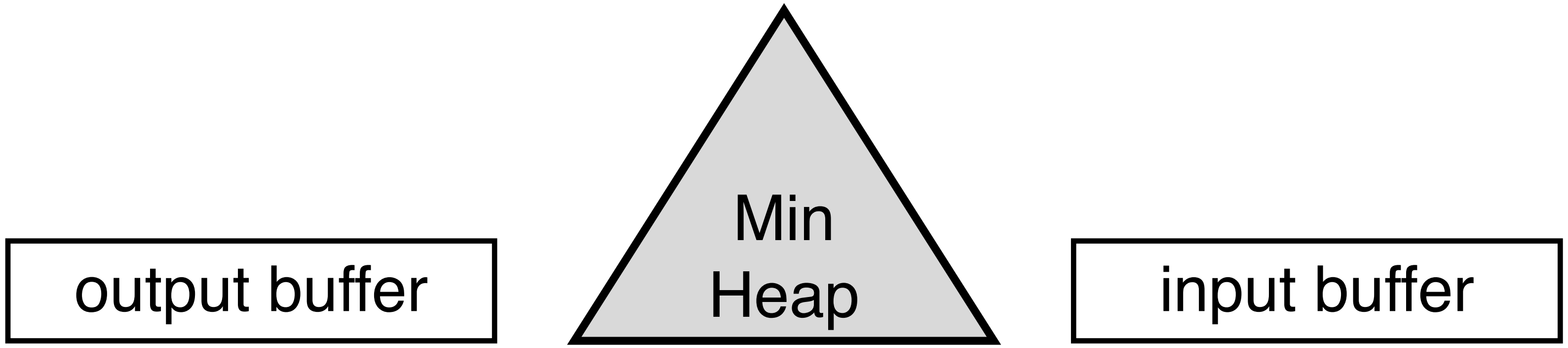


Naïve solution:

- Load | M | blocks
- Sort
- Output | M | blocks

Each run contains | M | blocks

Sort Merge Join – Replacement Selection

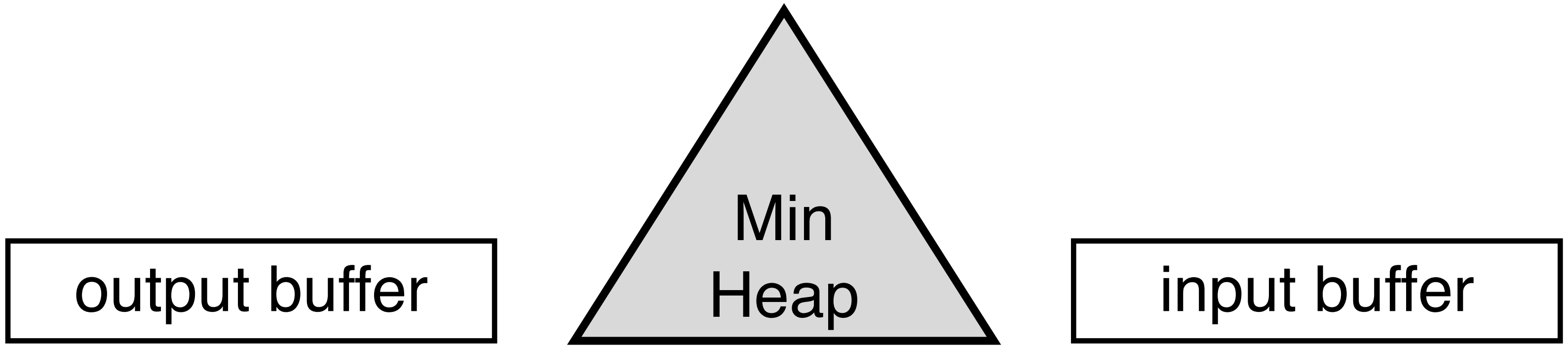


Replacement selection:

- load | M | blocks and sort

```
While heap is not empty
    Output one tuple and load one tuple from input buffer
    If the new tuple < any tuple in output
        save the tuple for next run (heap size reduces)
    else
        heap reorder
```

Sort Merge Join – Replacement Selection



Replacement selection:

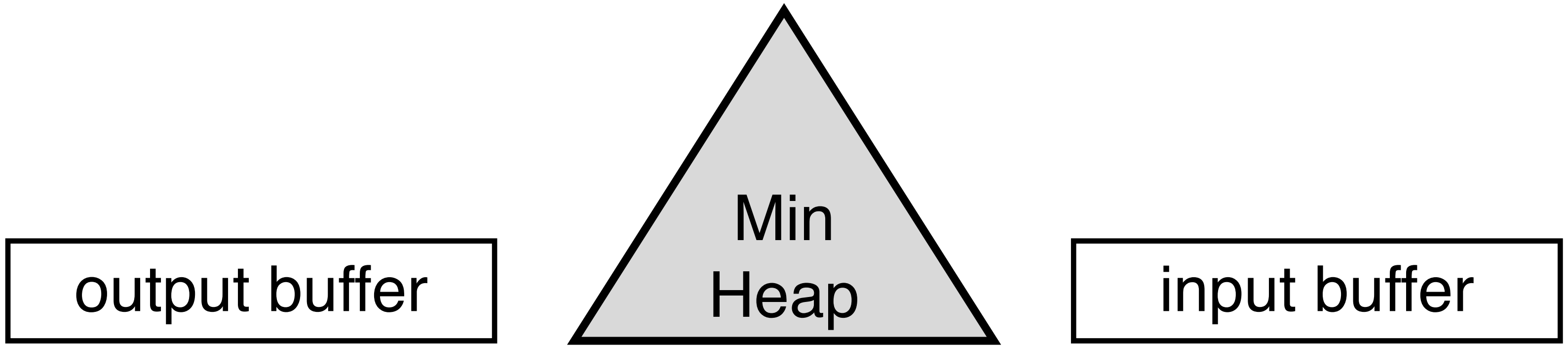
- load $|M|$ blocks and sort

Each run contains $2 \times |M|$ blocks

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ExternalSort.html

```
While heap is not empty
    Output one tuple and load one tuple from input buffer
    If the new tuple < any tuple in output
        save the tuple for next run (heap size reduces)
    else
        heap reorder
```

Sort Merge Join – Replacement Selection



Replacement selection:

- load | M | blocks and sort

Each run contains 2 × | M | blocks

https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/ExternalSort.html

```
While heap is not empty
    Output one tuple and load one tuple from input buffer
    If the new tuple < any tuple in output
        save the tuple for next run (heap size reduces)
    else
        heap reorder
```

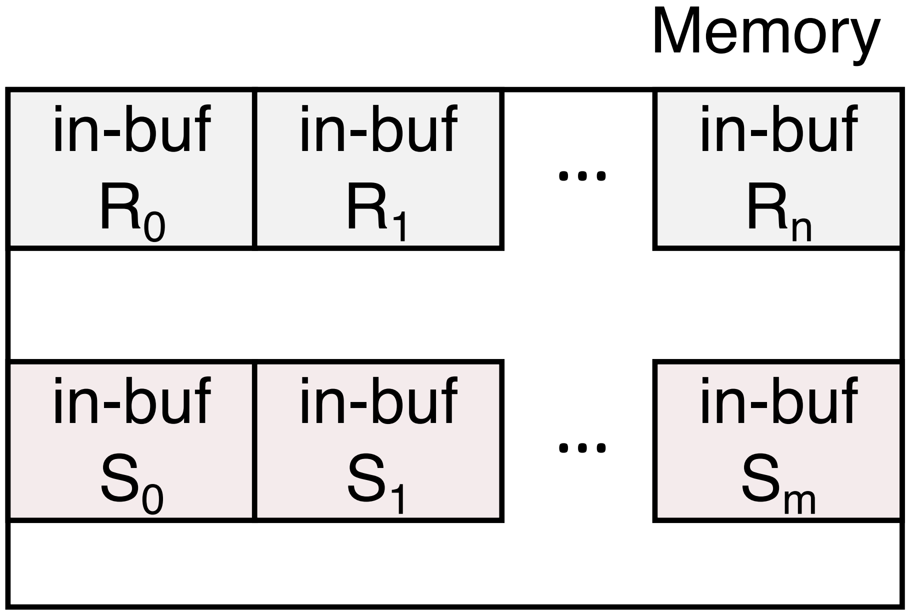
Total number of runs

$$= \frac{|S|}{2 \times |M|} + \frac{|R|}{2 \times |M|} \leq \frac{|S|}{|M|}$$

Sort Merge Join – Phase 2

Phase 2: Merge runs of S and R, output join result

- One input buffer required for each run



Memory layout in Phase 2

Sort Merge Join – Phase 2

Phase 2: Merge runs of S and R, output join result

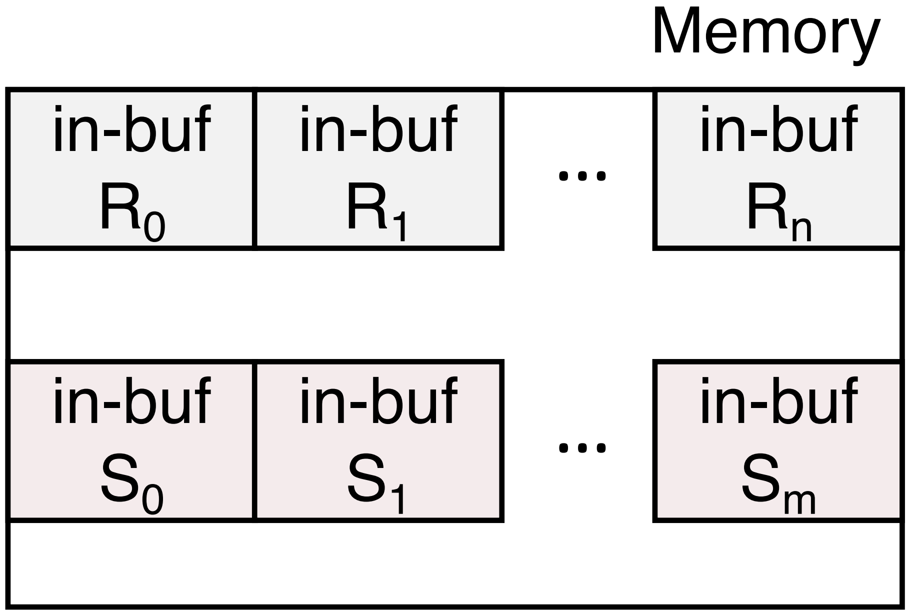
- One input buffer required for each run

Requirement

$$| M | \geq \text{total number runs}$$

Satisfied if $\quad | M | \geq \dfrac{| S |}{| M |}$

namely $\quad | M | \geq \sqrt{| S |}$



Memory layout in Phase 2

Hash Join

Build a hash table on the smaller relation (**R**) and probe with larger (**S**)

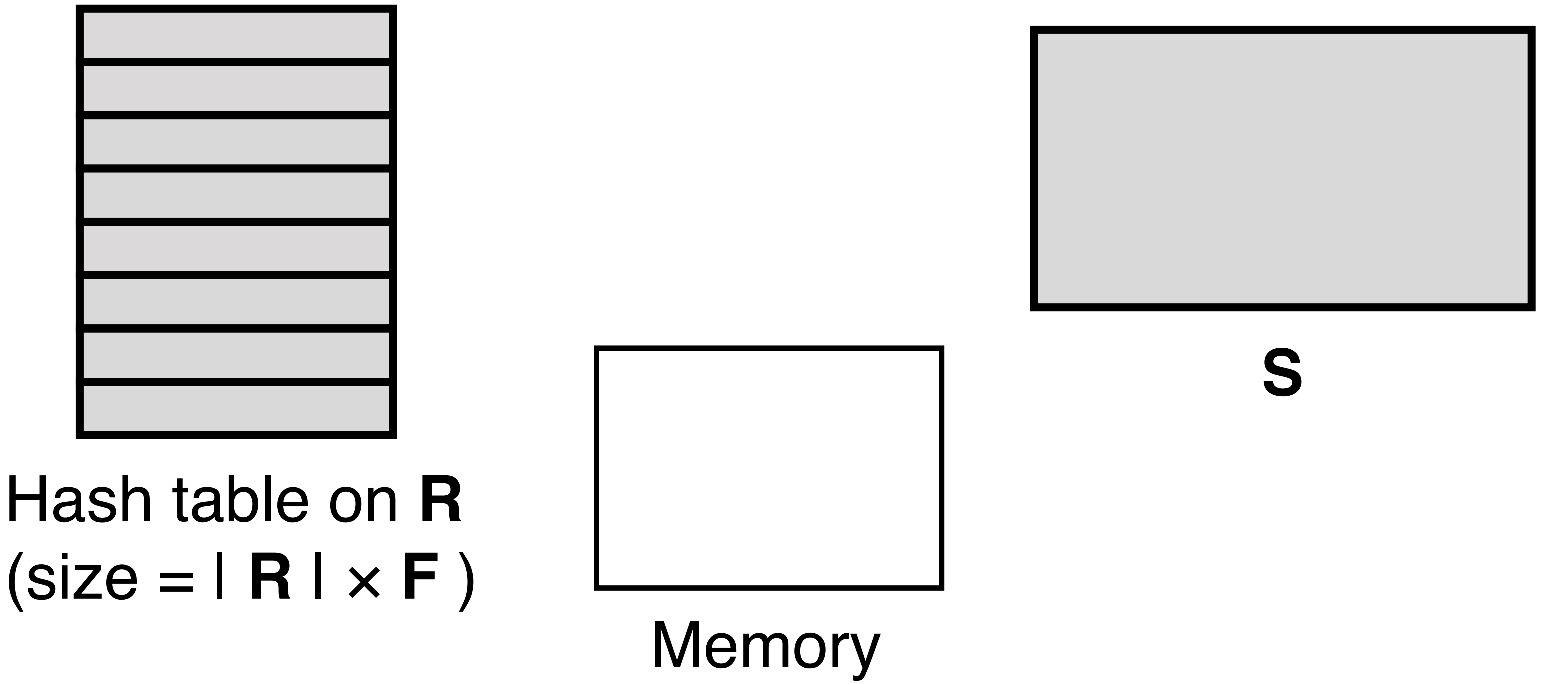
Hash tables have overhead, call it **F**

When **R** doesn't fit fully in memory, partition hash space into ranges

Hash table on **R**
(size = | **R** | × **F**)

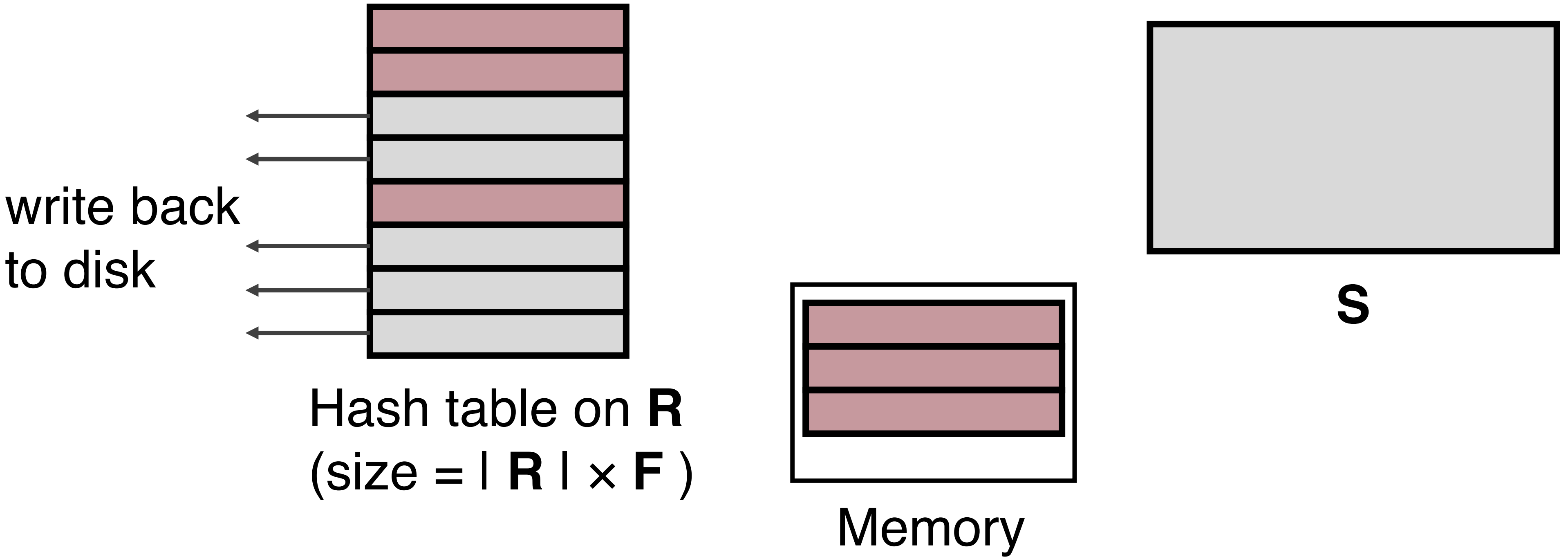
Simple Hash Join

- Build a hash table on **R**



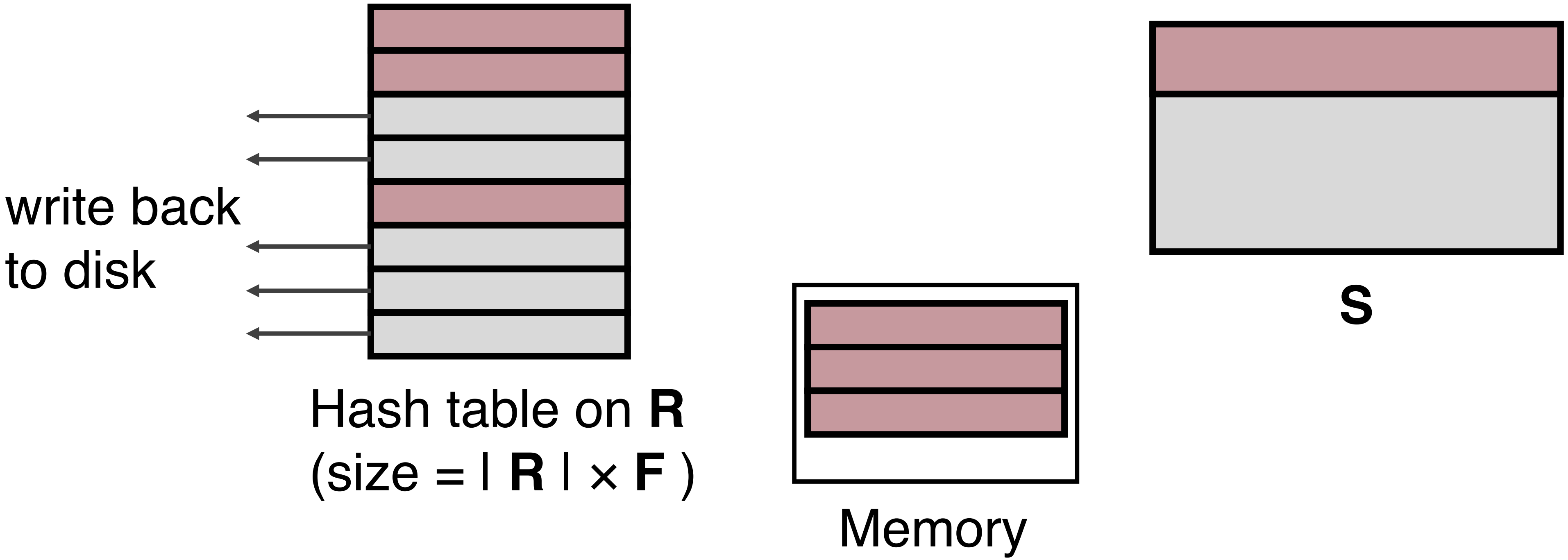
Simple Hash Join – 1st pass

- Build a hash table on **R**
- If **R** does not fit in memory, find a subset of buckets that fit in memory



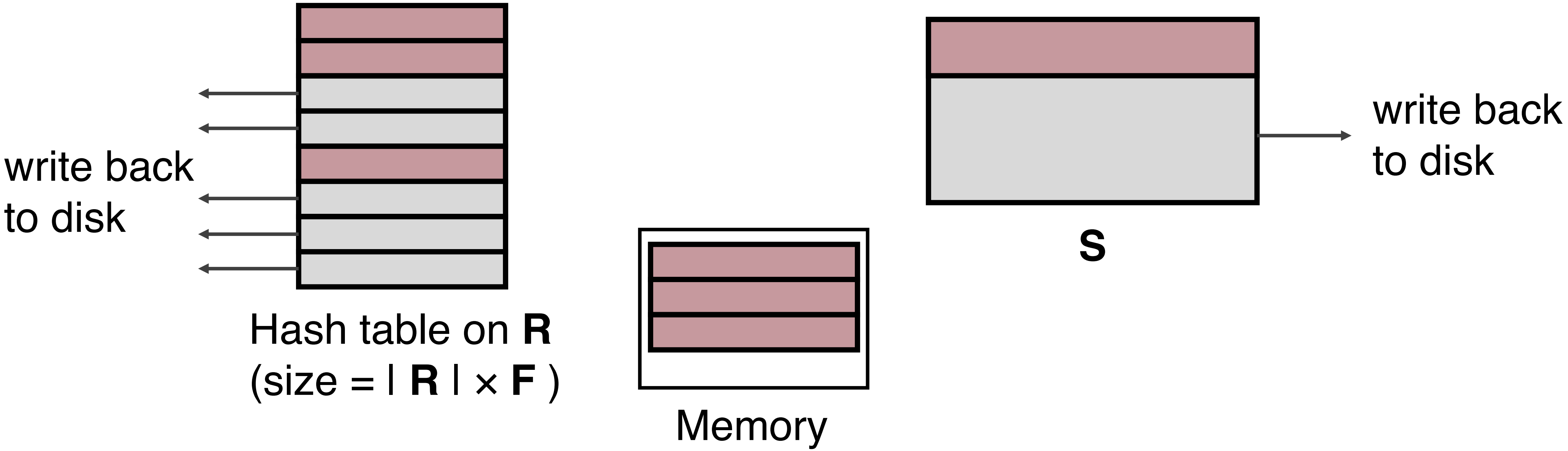
Simple Hash Join – 1st pass

- Build a hash table on **R**
- If **R** does not fit in memory, find a subset of buckets that fit in memory
- Read in **S** to join with the subset of **R**



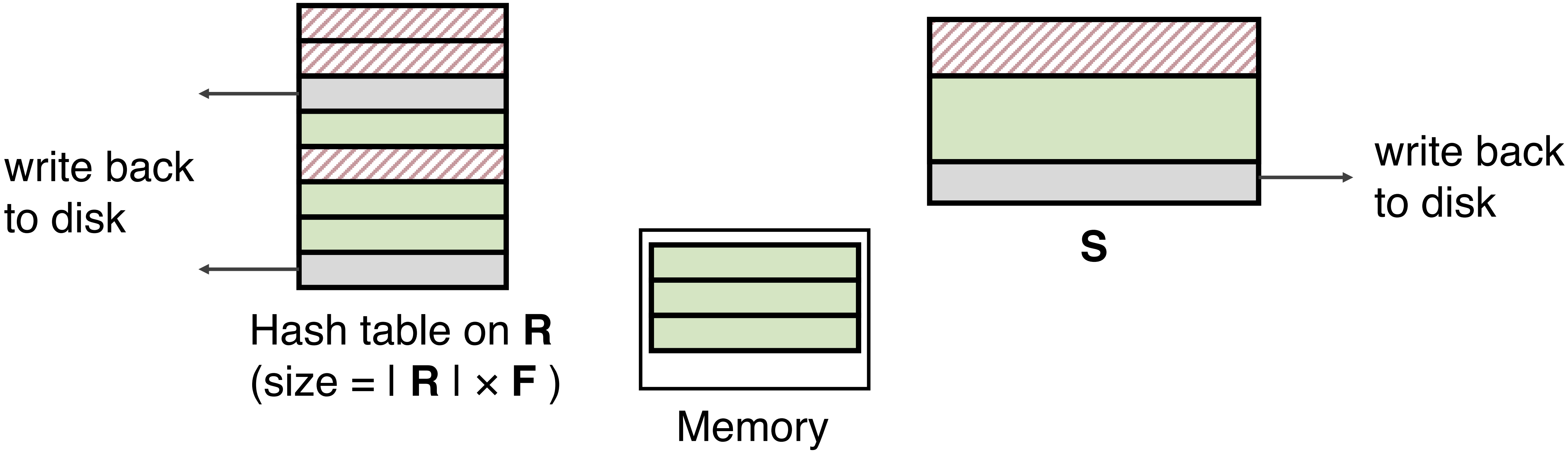
Simple Hash Join – 1st pass

- Build a hash table on **R**
- If **R** does not fit in memory, find a subset of buckets that fit in memory
- Read in **S** to join with the subset of **R**
- The remaining tuples of **S** and **R** are written back to disk



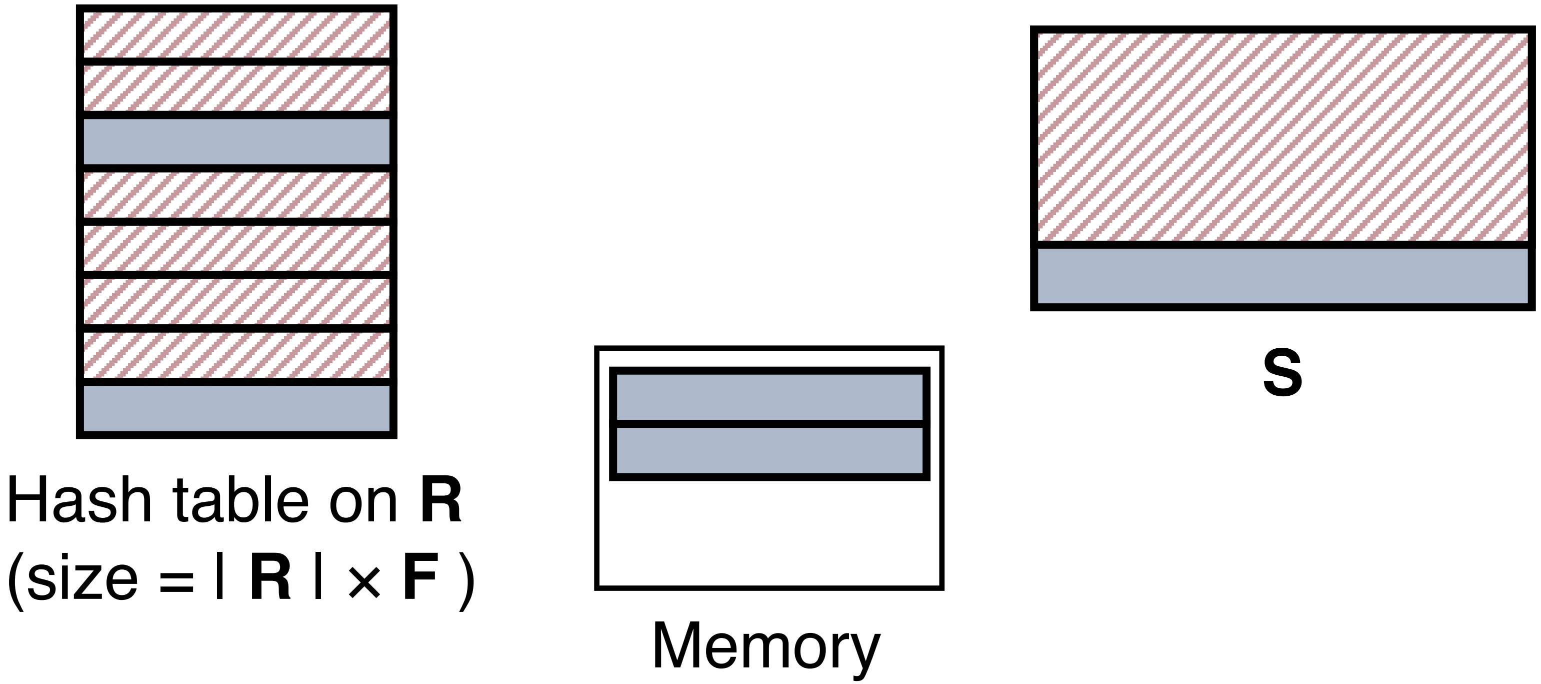
Simple Hash Join – 2^{nd} pass

- Build a hash table on **R**
- If **R** does not fit in memory, find a subset of buckets that fit in memory
- Read in **S** to join with the subset of **R**
- The remaining tuples of **S** and **R** are written back to disk



Simple Hash Join – 3rd pass

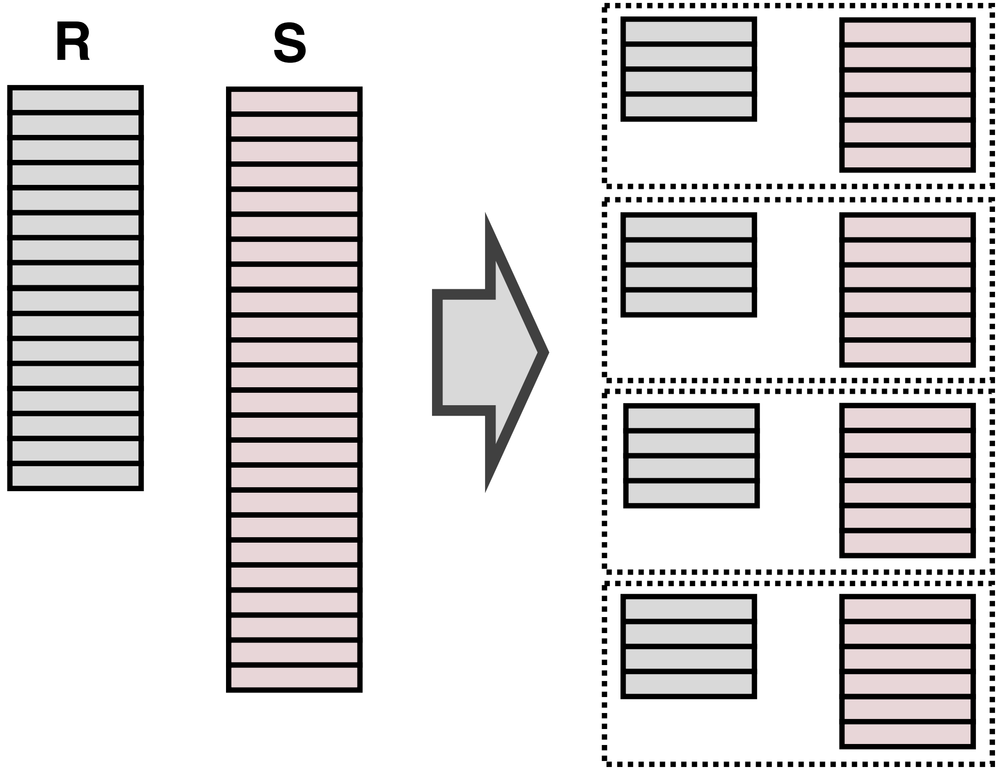
- Build a hash table on **R**
- If **R** does not fit in memory, find a subset of buckets that fit in memory
- Read in **S** to join with the subset of **R**
- The remaining tuples of **S** and **R** are written back to disk



GRACE Hash Join

Phase 1: Partition both R and S into pairs of shards

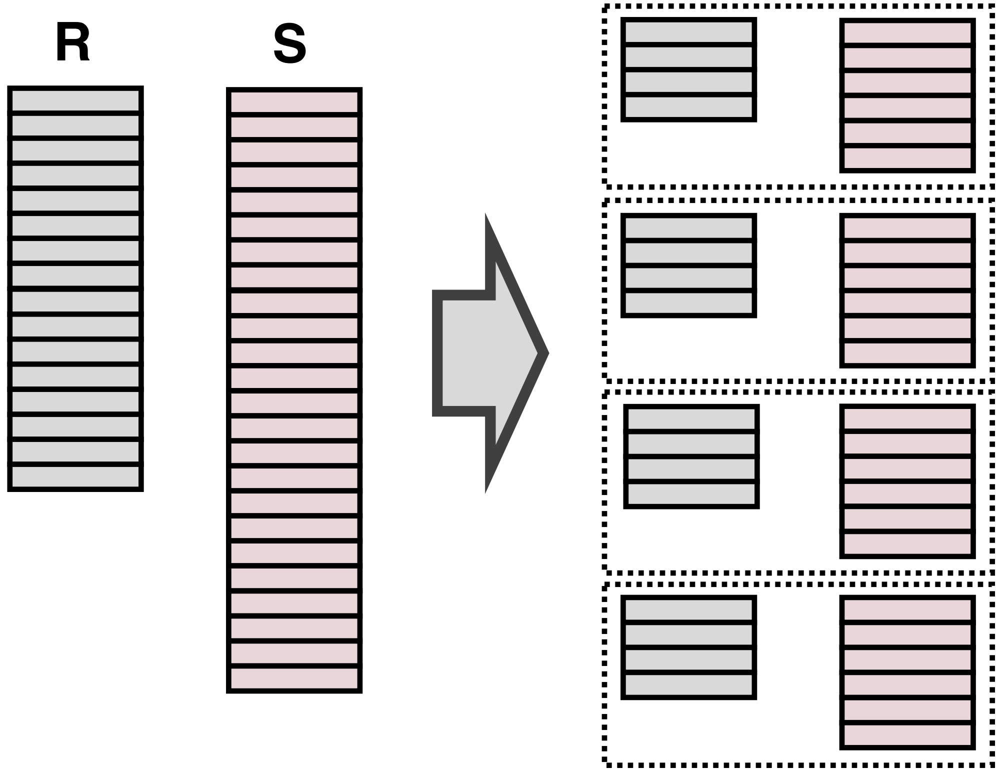
Phase 2: Separately join each pairs of partitions

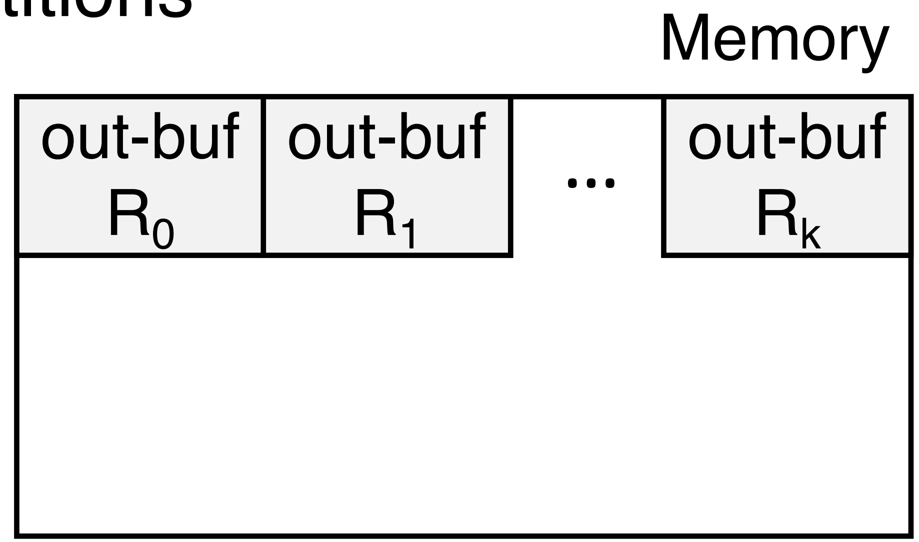


GRACE Hash Join

Phase 1: Partition both R and S into pairs of shards

Phase 2: Separately join each pairs of partitions



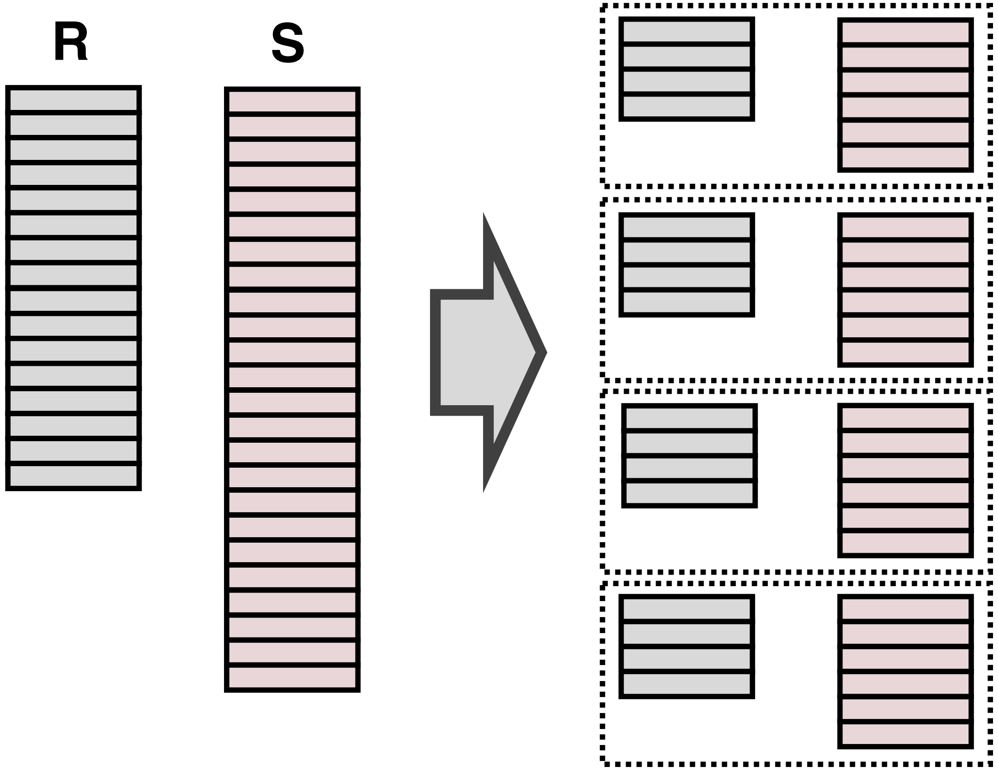


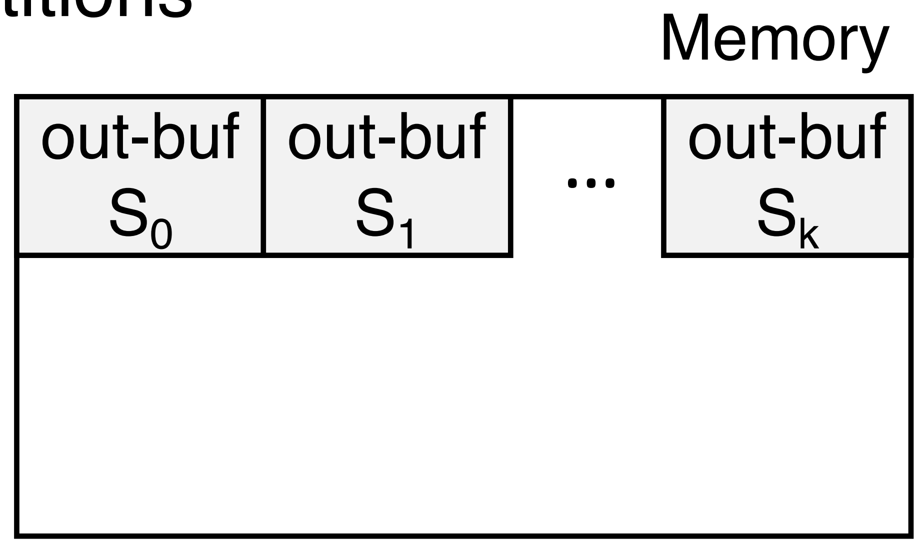
Memory layout in Phase 1

GRACE Hash Join

Phase 1: Partition both R and S into pairs of shards

Phase 2: Separately join each pairs of partitions



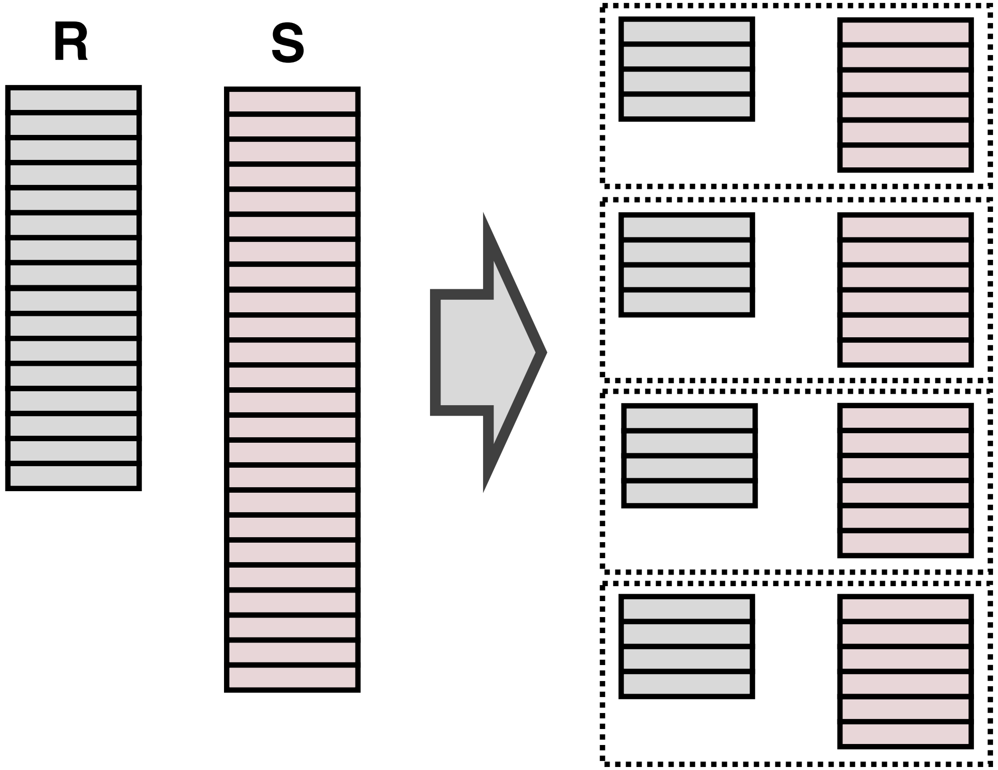


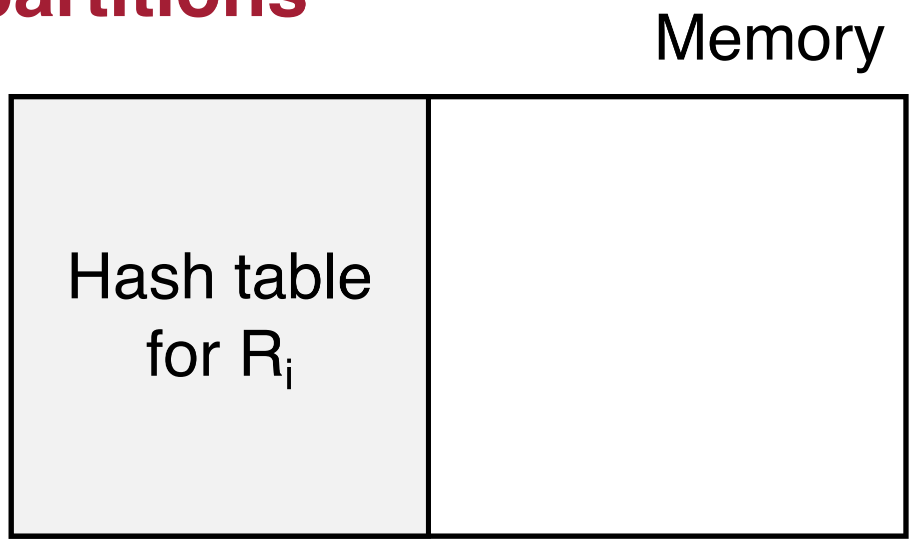
Memory layout in Phase 1

GRACE Hash Join

Phase 1: Partition both R and S into pairs of shards

Phase 2: Separately join each pairs of partitions





Memory layout in Phase 2

GRACE Hash Join

Assume **k** partitions for **R** and **S**

In phase 1, needs one output buffer (i.e., block) for each partition

$$k \leq |M|$$

GRACE Hash Join

Assume **k** partitions for **R** and **S**

In phase 1, needs one output buffer (i.e., block) for each partition

$$k \leq |M|$$

In phase 2, the hash table of each shard of **R** must fit in memory

$$\frac{|R|}{k} \times F \leq |M|$$

GRACE Hash Join

Assume **k** partitions for **R** and **S**

In phase 1, needs one output buffer (i.e., block) for each partition

$$k \leq |M|$$

In phase 2, the hash table of each shard of **R** must fit in memory

$$\frac{|R|}{k} \times F \leq |M|$$

The maximum size of **R** to perform Grace hash join:

$$|R| \leq \frac{|M|}{F} k \leq \frac{|M|^2}{F} \qquad |M| \geq \sqrt{|R| \times F}$$

GRACE vs. Simple Hash Join

When $| R | \times F < | M |$

- Simple hash join incurs no IO traffic (better)
- GRACE hash join writes and reads each table once

When $| M |^2 \geq | R | \times F >> | M |$

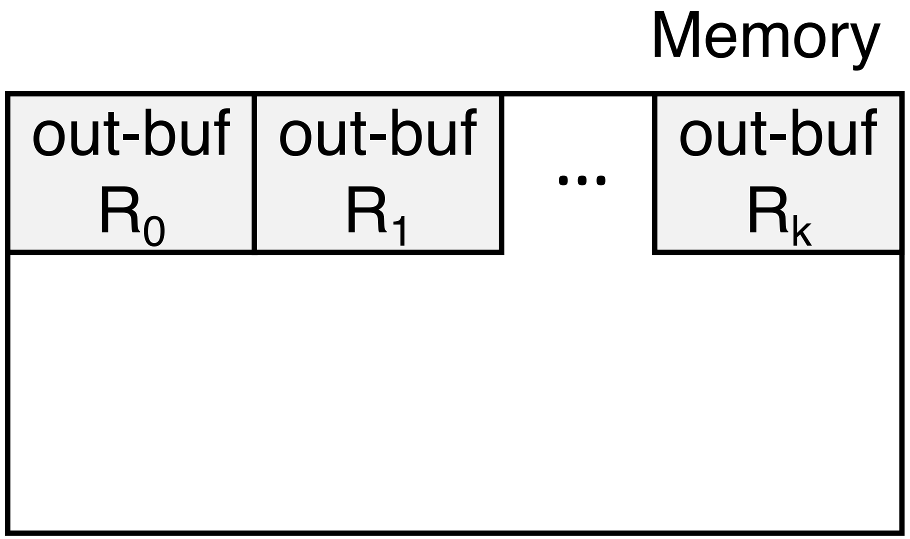
- Simple hash join incurs significant IO traffic
- GRACE hash join writes and reads each table once (better)

Hybrid Hash Join

When you have two algorithms that are good in different settings, create a hybrid!

Hybrid Hash Join

When you have two algorithms that are good in different settings, create a hybrid!



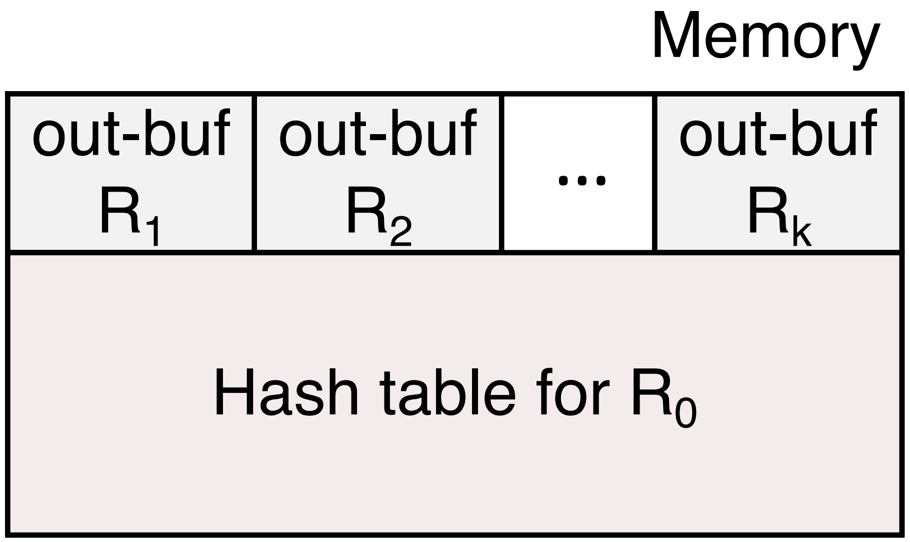
Memory layout in Phase 1 of GRACE hash join

Hybrid Hash Join

When you have two algorithms that are good in different settings, create a hybrid!

For example

- If | R | = 2 * | M |
- R needs to be partitioned into only 2 shards
- Only 2 out-bufs are required for partitioning
- Rest of memory can be used to build hash table for R to avoid writing some of R to disk

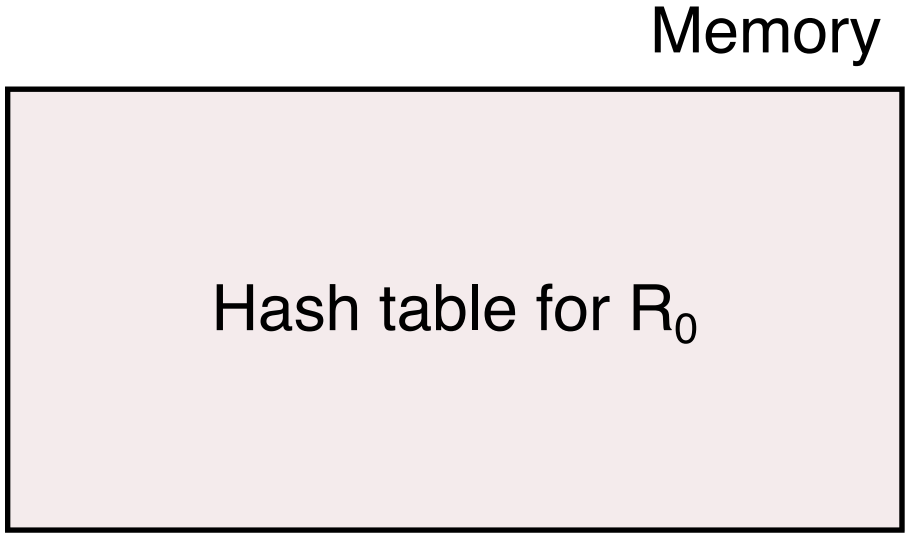


Memory layout in Phase 1 of hybrid hash join

Hybrid Hash Join

Case 1: $|R| \times F < |M|$

- No need to partition R
- Identical to simple hash join



Memory layout in Phase 1 of hybrid hash join

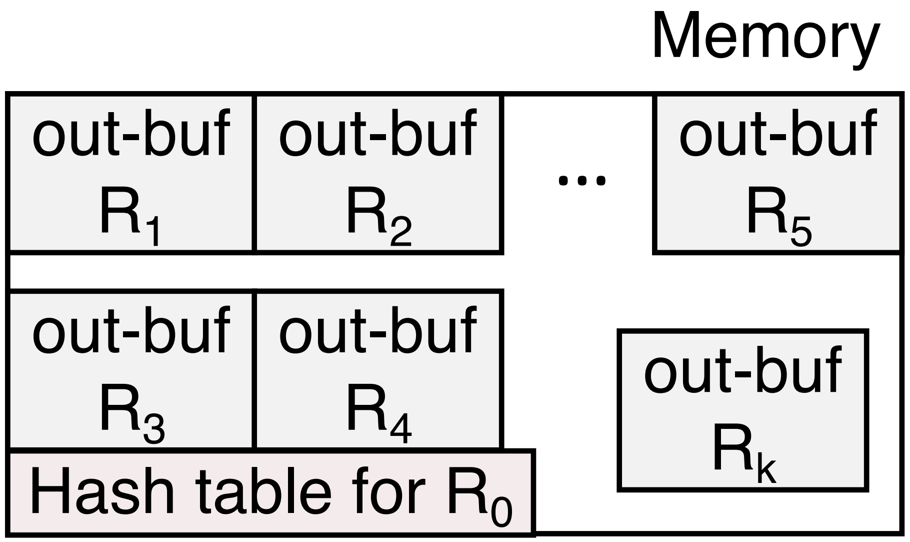
Hybrid Hash Join

Case 1: | R | × F < | M |

- No need to partition R
- Identical to simple hash join

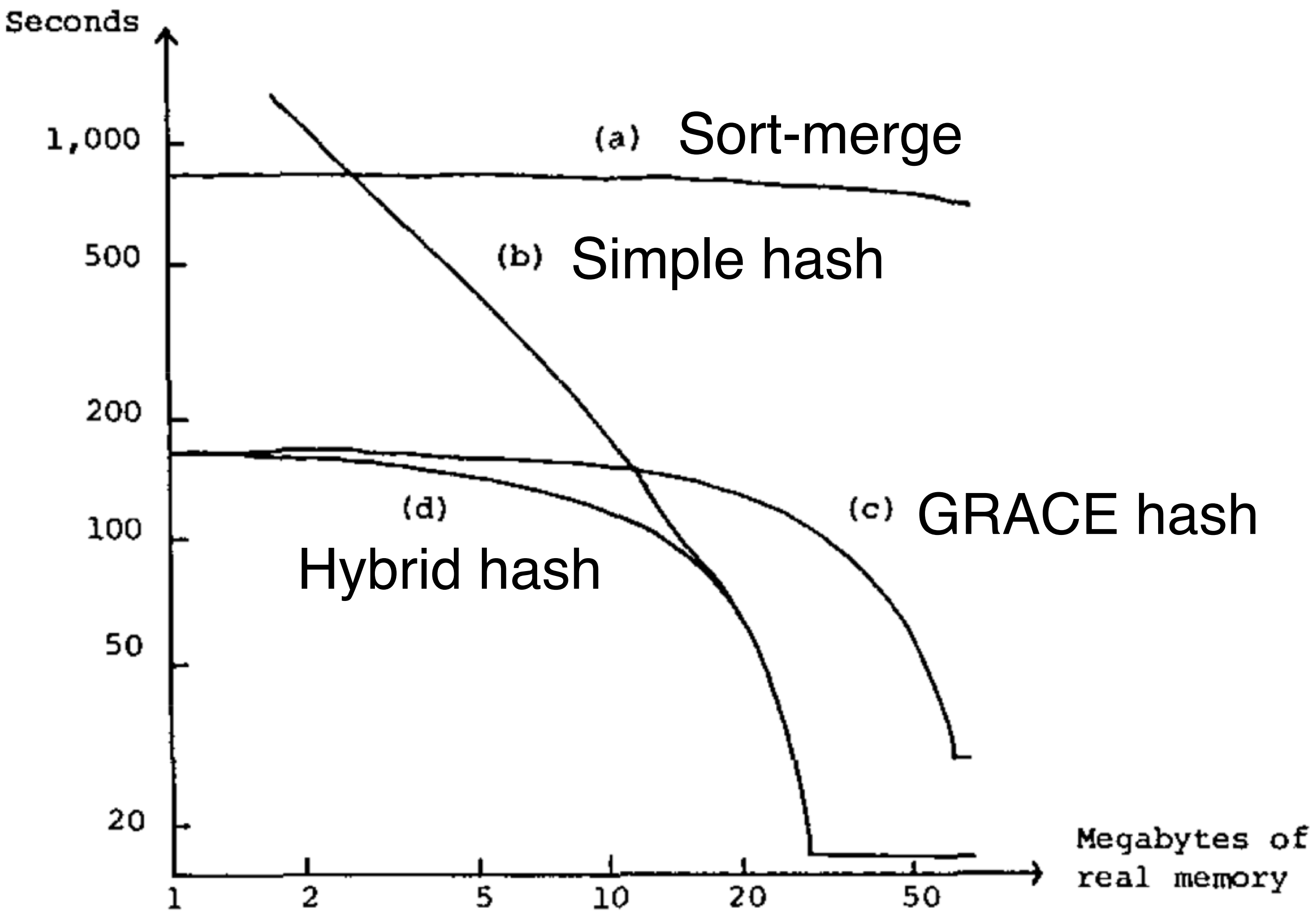
Case 2: | R | × F >> | M |

- Need
- Similar to GRACE hash join



Memory layout in Phase 1 of hybrid hash join

Evaluation



- Conclusion 1: Hash join is generally better than sort-merge join
- Conclusion 2: Hybrid hash join is strictly better than simple and GRACE hash joins

Partition Overflow

So far we assume uniform random distribution for **R** and **S**

What if we guess wrong on size required for R hash table and a partition does not fit in memory?

Solution: further divide into smaller partitions range

Additional Techniques

Babb array (or bitmap filter)

- Set a bit for each R tuple
- Use to filter S during initial scan, discard tuple if missing in array

Semijoin

- Project join attributes from R, join to S, then join that result back to R
- Useful if full R tuples won't fit into memory, but join will be selective and filter many S tuples
- Can be added to any join algorithm above

Join – Comments and Q/A

- Lack of experiments
- Conclusions still hold for modern systems?
- With duplicate join keys, a partition may never be smaller than memory size
- Why is a run $2 \times | M |$ long?
- Hash vs. Merge for already sorted data
- Join in a distributed system?
- Is the math/proof important?
- Multiple joins? non-equijoin?

Group Discussion

In a modern in-memory DBMS, the entire database fits in DRAM. In such a system, can similar optimizations be applied based on the performance gap between on-chip SRAM caches vs. DRAM? Please discuss the opportunities and challenges of this approach.

Before Next Lecture

Submit review for

Peter Boncz, et al., Database Architecture Optimized for the new Bottleneck: Memory Access. VLDB, 1999