WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems

Lecture 2: Join

Xiangyao Yu

9/9/2025

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 notations

Join algorithms

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

Partition overflow and additional techniques

Agenda

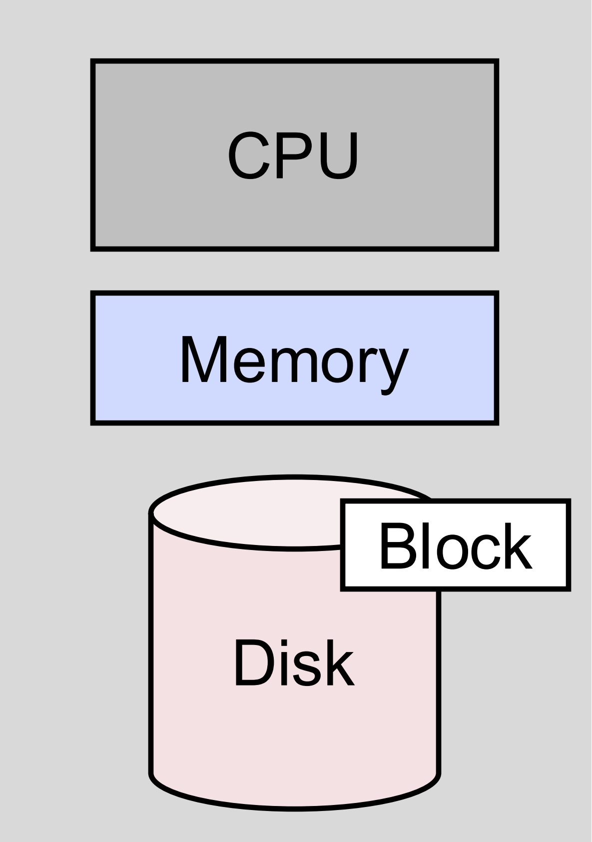
System architecture and notations

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

- No multi-core synchronization complexity
- Could be built on systems of the day

Memory

- Tens of Megabytes
- Good for **both sequential and random** accesses
- Capacity is smaller than disk

Disk

- Good for **only sequential** accesses

Notation

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

Join: S ⋈ R

Memory: M

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

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

Focus only on equi-join

Notation

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

Join: S ⋈ R

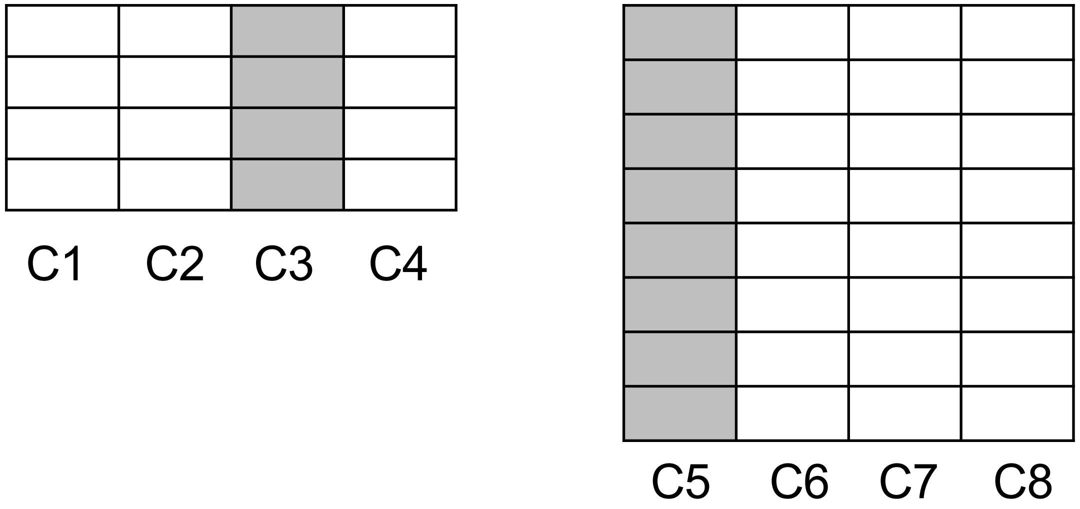
Memory: M

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

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

Relation R

Relation S



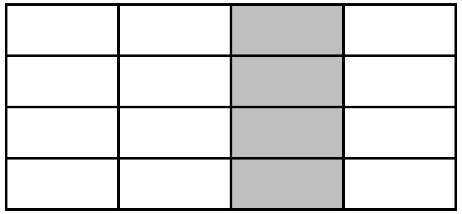

```
SELECT *
FROM R, S
WHERE R.C3 = S.C5
```

Notation

Vanilla query executor

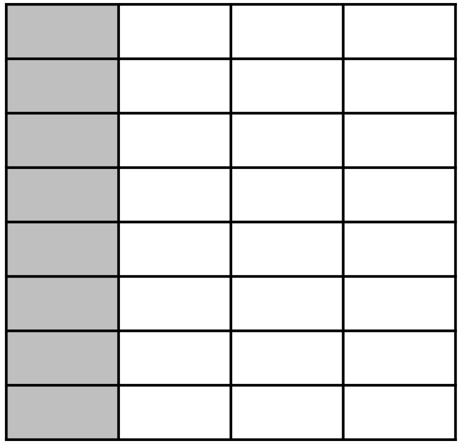
```
answer = {}
for t1 in R do
  for t2 in S do
    if R.C3 = S.C5
      then answer = answer ∪ {(C1,…,C8)}
return answer
```

Relation R



C1 C2 C3 C4

Relation S



C5 C6 C7 C8

```
SELECT *
FROM R, S
WHERE R.C3 = S.C5
```

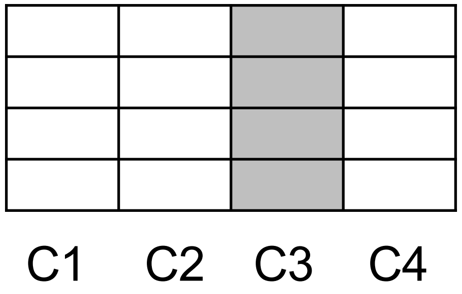
Notation

```
answer = {}
for t1 in R do
  for t2 in S do
    if R.C3 = S.C5
      then answer = answer U {(C1,…,C8)}
return answer
```

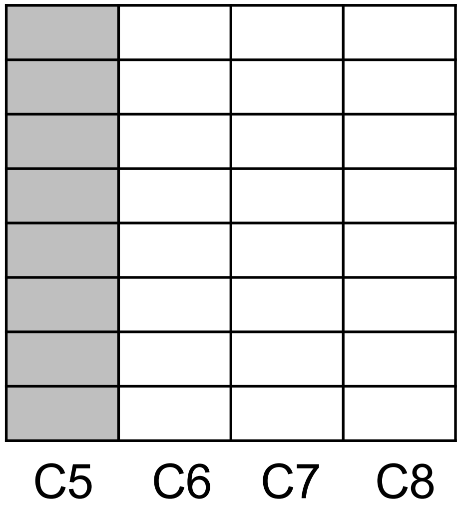
Vanilla query executor

Key question: How to execute a join fast?

Relation R



Relation S




```
SELECT *
FROM R, S
WHERE R.C3 = S.C5
```

Agenda

System architecture and notations

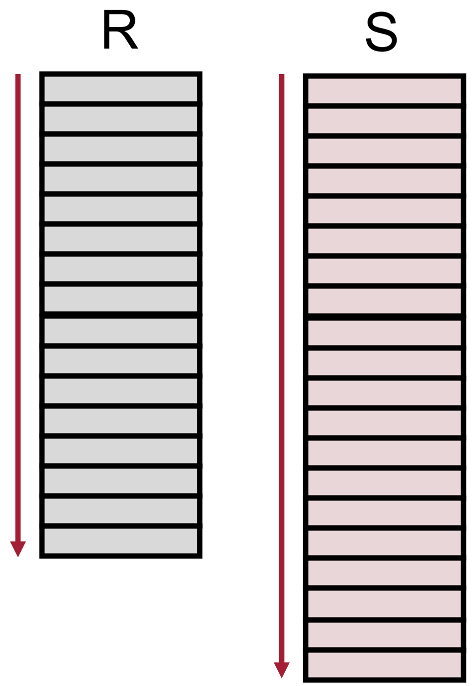
Join algorithms

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

Partition overflow and additional techniques

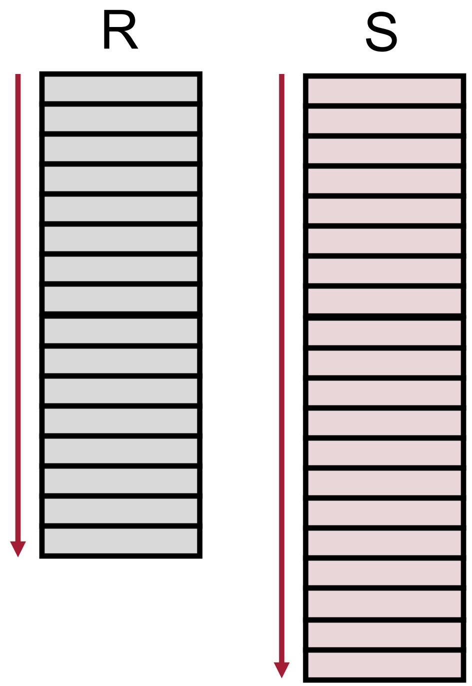
Sort Merge Join

Key idea: sort both relations based on join attributes, then traverse both relations in the sorting order



Sort Merge Join

Key idea: sort both relations based on join attributes, then traverse both relations in the sorting order

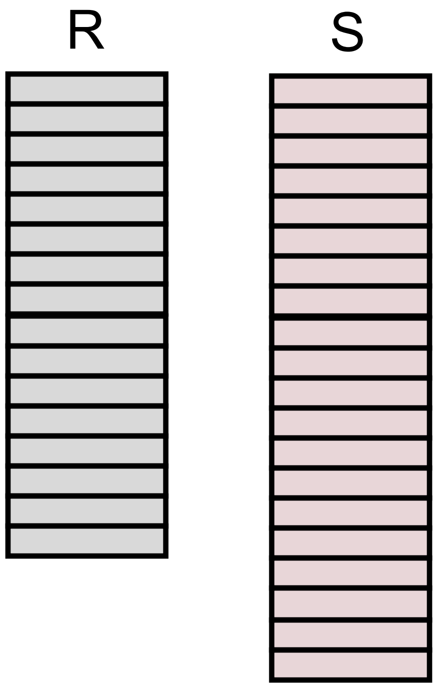


Challenge: If a relation does not fit in memory, need to sort data on disk

Sort Merge Join

Phase 1: Produce sorted runs of S and R

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

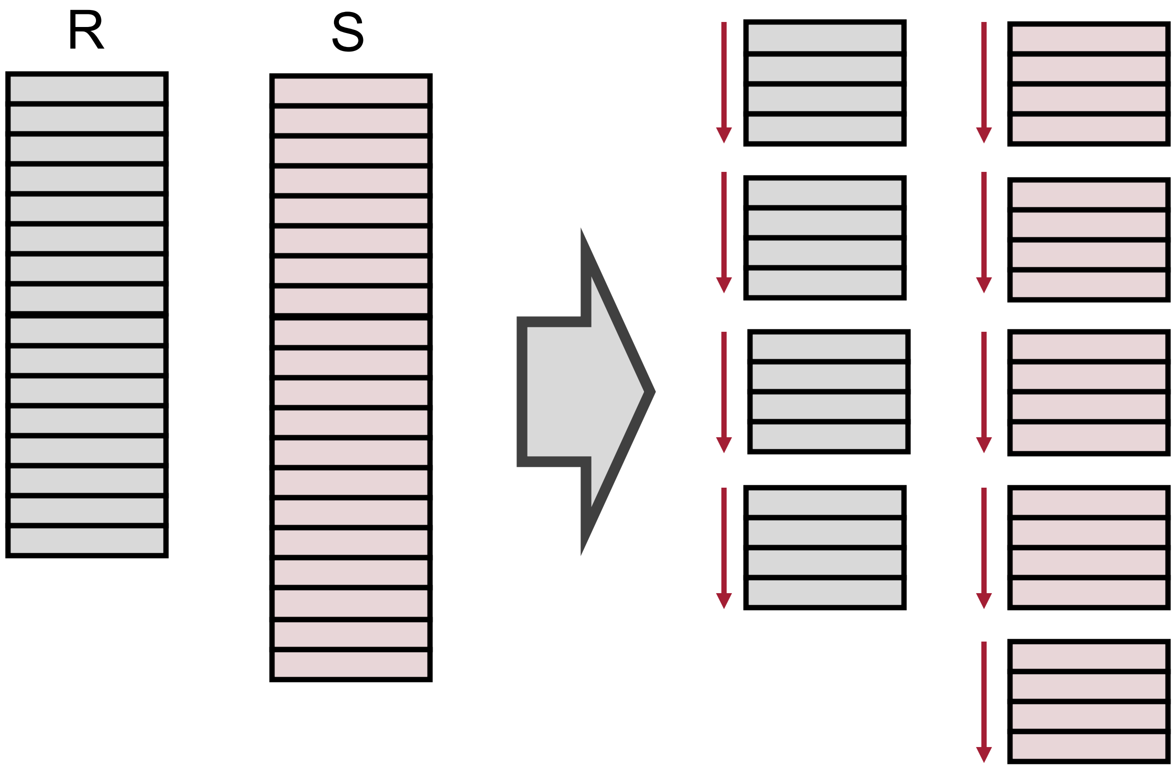


Unsorted R and S

Sort Merge Join

Phase 1: Produce sorted runs of S and R

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



Each sorted run can fit in memory

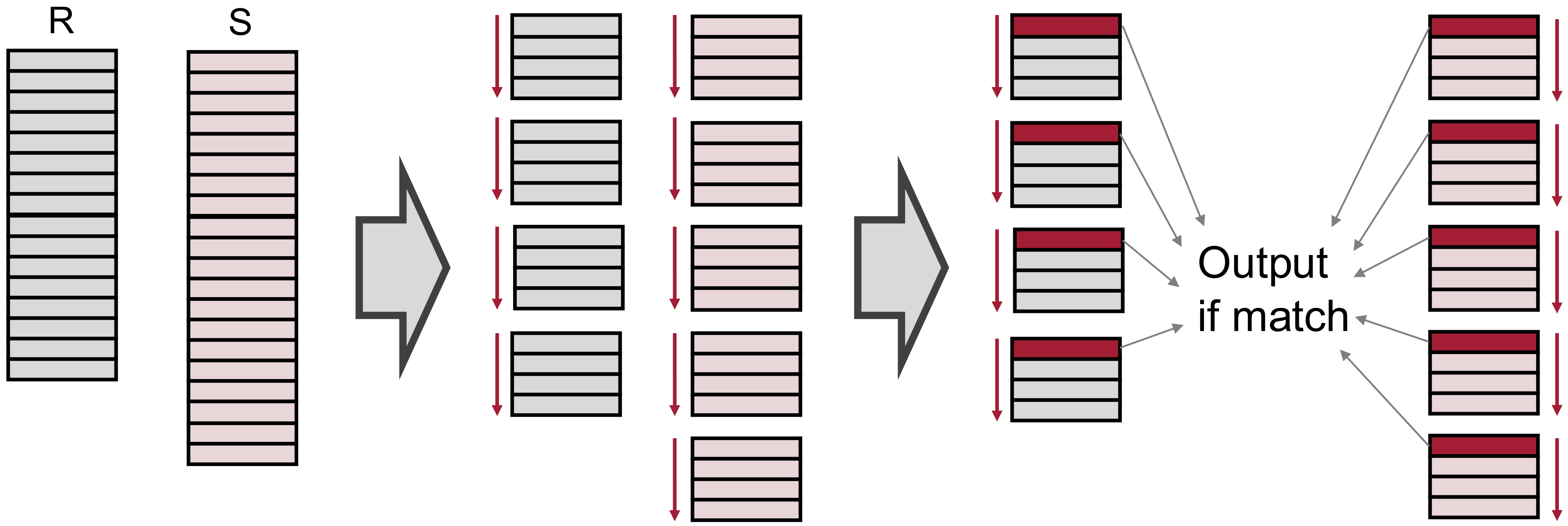
Unsorted R and S

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

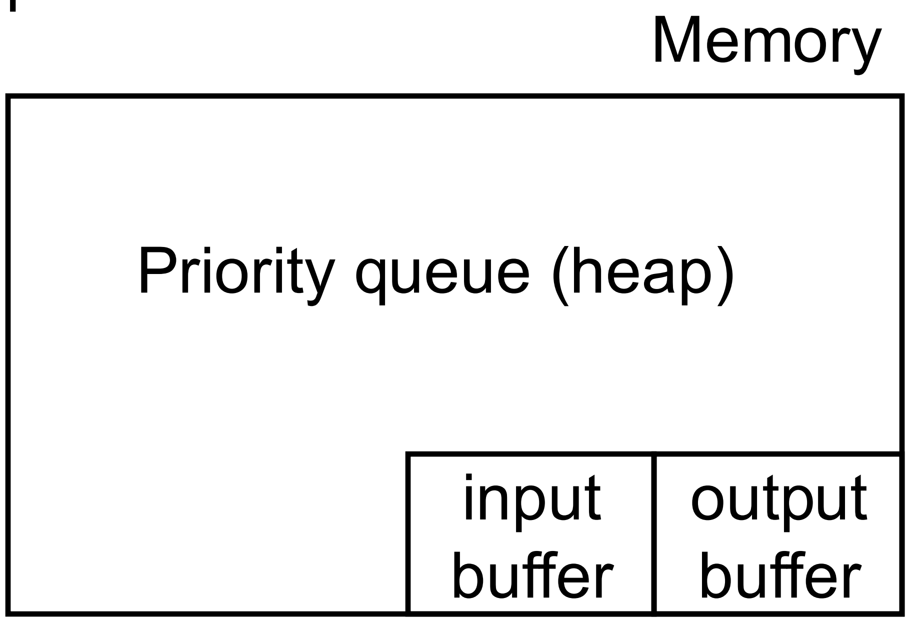


Unsorted R and S   Sorted runs of R and S   Find matches in sorted runs

Sort Merge Join – Phase 1

Phase 1: Produce sorted runs of S and R

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



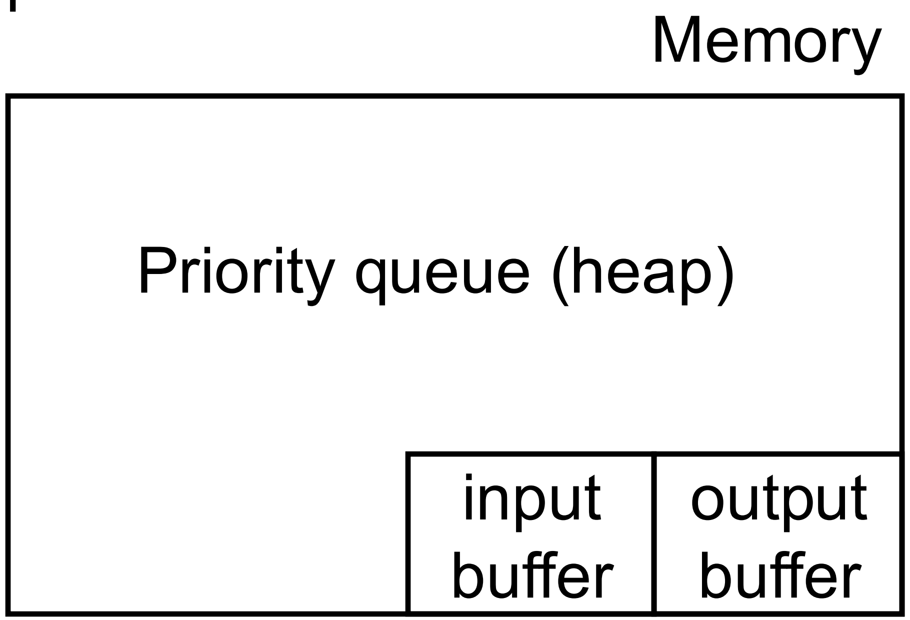
Memory layout in Phase 1

Sort Merge Join – Phase 1

Phase 1: Produce sorted runs of S and R

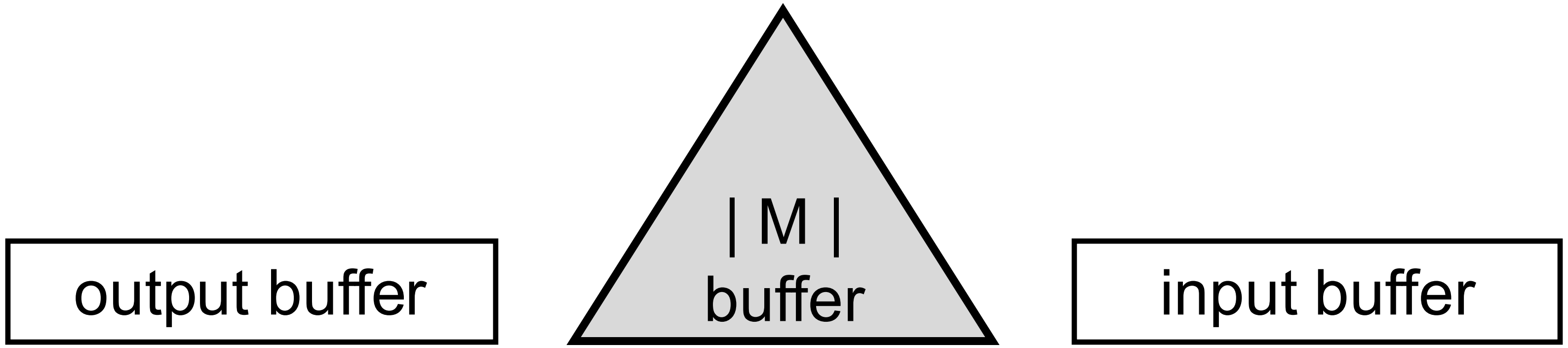
- Each run of S will be **2 × | M |** average length

Q: Where does 2 come from?
A: Replacement selection



Memory layout in Phase 1

Sort Merge Join – Replacement Selection

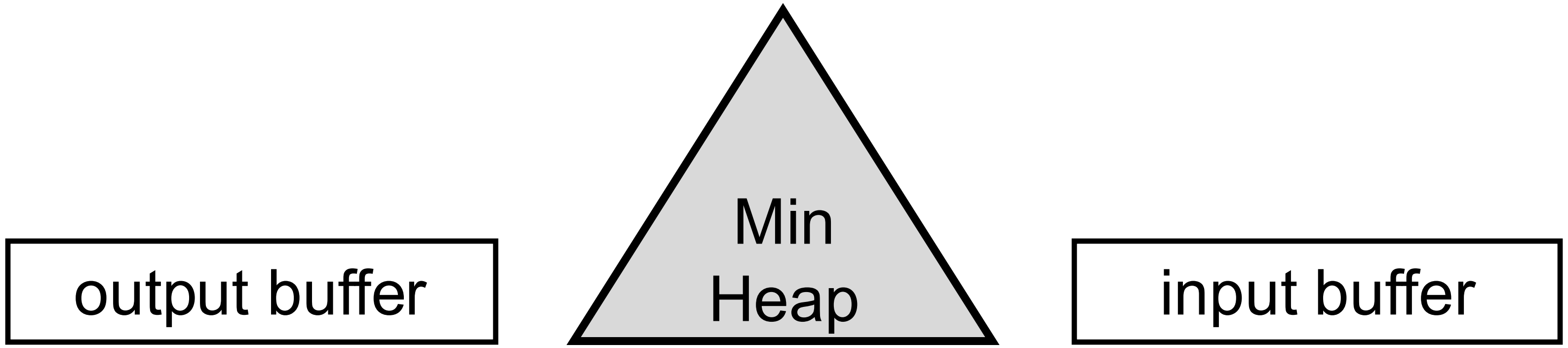


Naïve solution:

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

Each run contains | M | blocks

Sort Merge Join – Replacement Selection

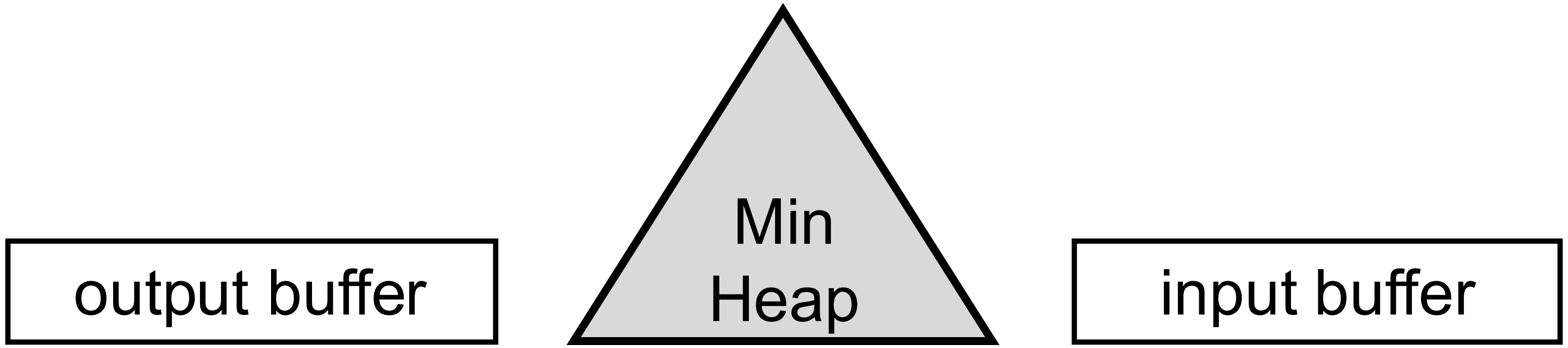


Replacement selection:

- load | M | blocks and sort

While heap is not empty
    If new tuple ≥ all tuples in output
        add new tuple to heap
    else
        save new tuple for next run

Sort Merge Join – Replacement Selection



Replacement selection:

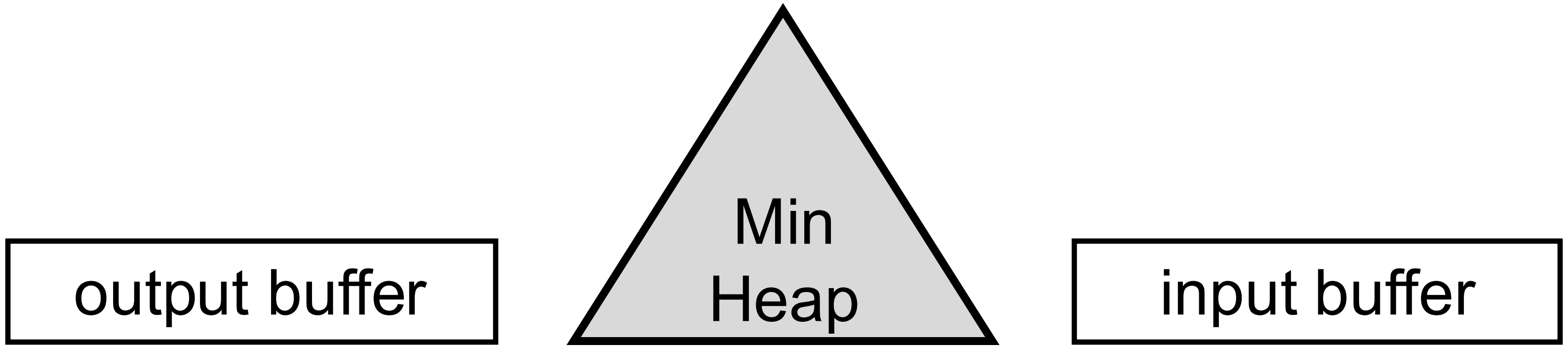
- load | M | blocks and sort

A run contains 2 × | M | blocks on average

While heap is not empty
    If new tuple ≥ all tuples in output
        add new tuple to heap
    else
        save new tuple for next run

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

Sort Merge Join – Replacement Selection



Replacement selection:

- load | M | blocks and sort

A run contains 2 × | M | blocks on average

While heap is not empty
    If new tuple ≥ all tuples in output
        add new tuple to heap
    else
        save new tuple for next run

Total number of runs

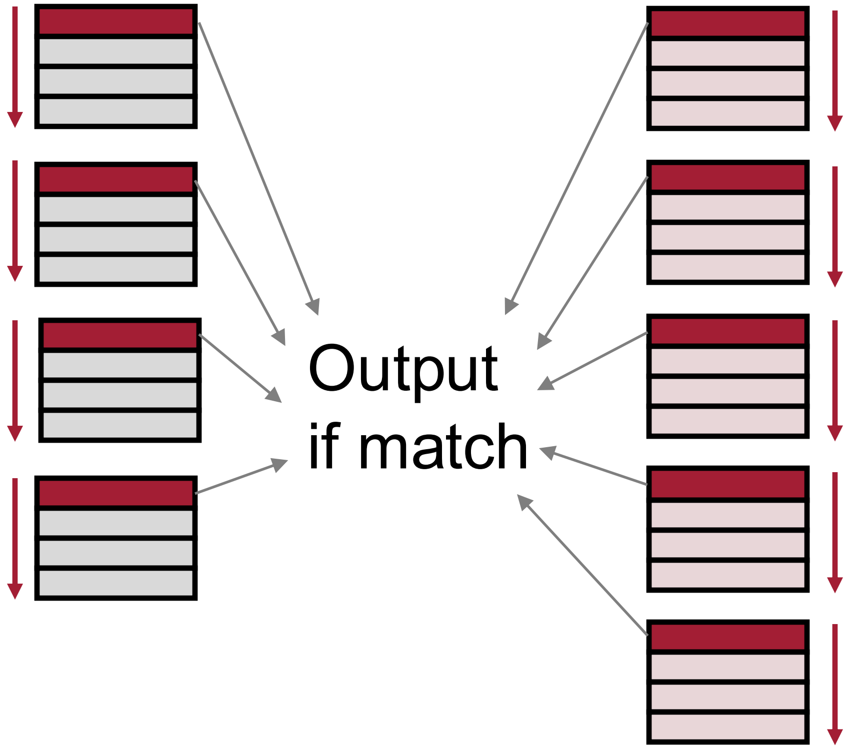
$$= \frac{|S|}{2 \times |M|} + \frac{|R|}{2 \times |M|} \leq \frac{|S|}{|M|}$$

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

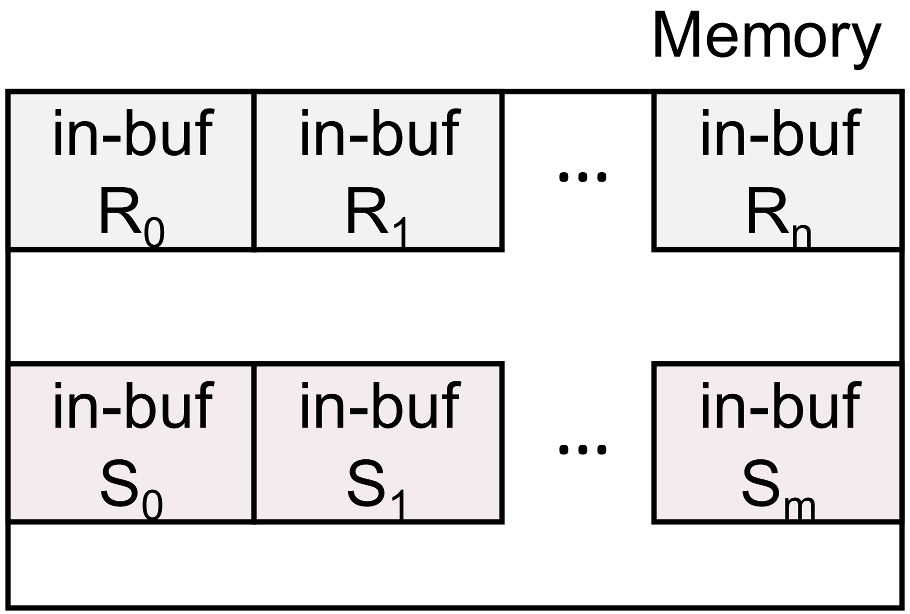
Sort Merge Join – Phase 2

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

- One input buffer required for each run



Find matches in sorted runs



Memory layout in Phase 2

Sort Merge Join – Phase 2

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

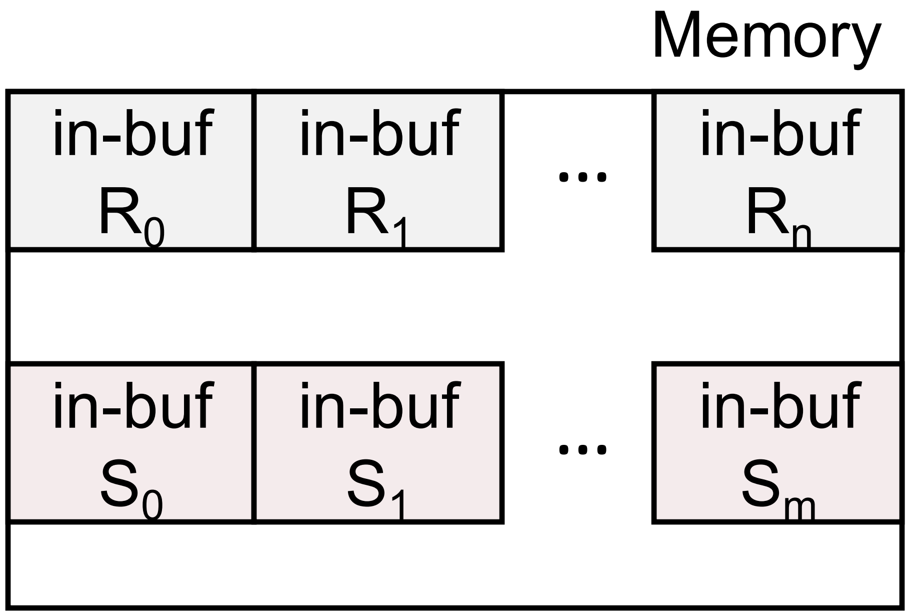
- One input buffer required for each run

Requirement

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

Satisfied if $| M | \geq \frac{| S |}{| M |}$

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



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**)

Agenda

System architecture and notations

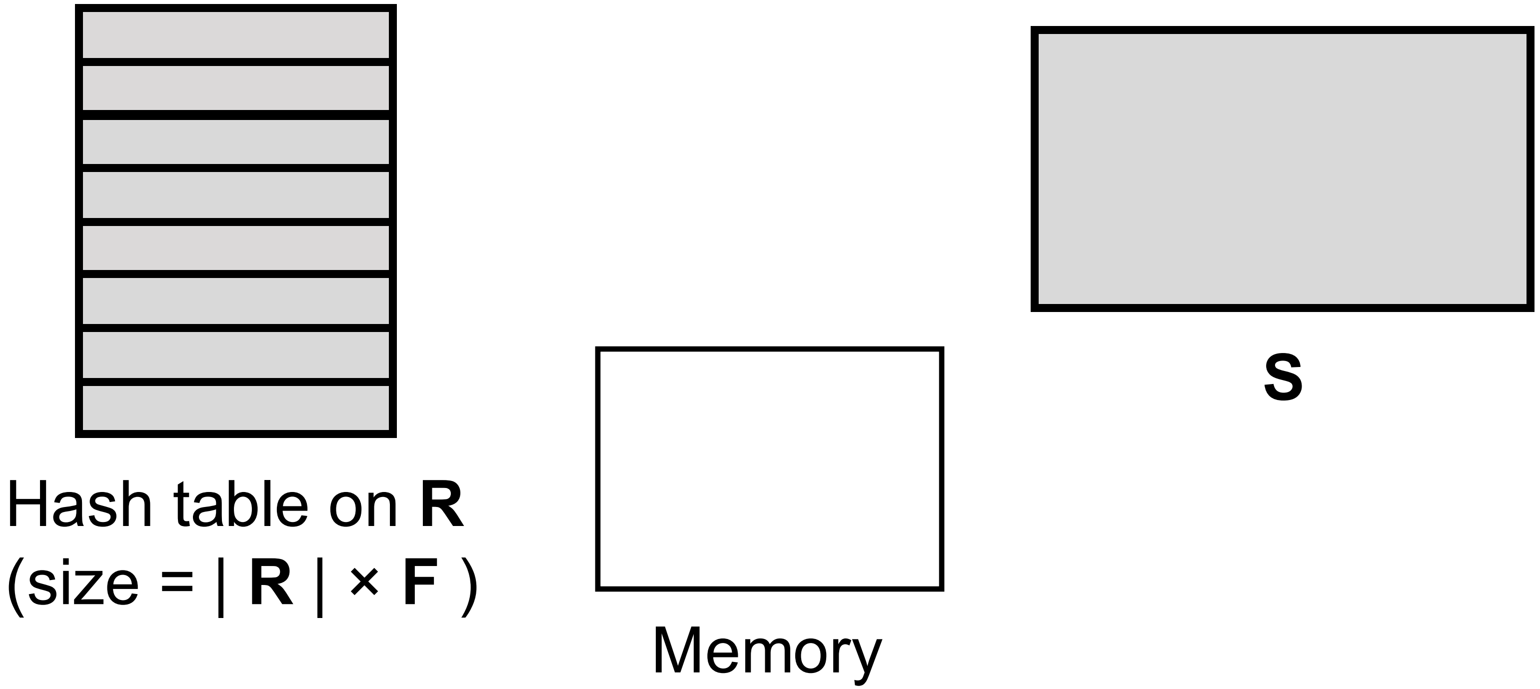
Join algorithms

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

Partition overflow and additional techniques

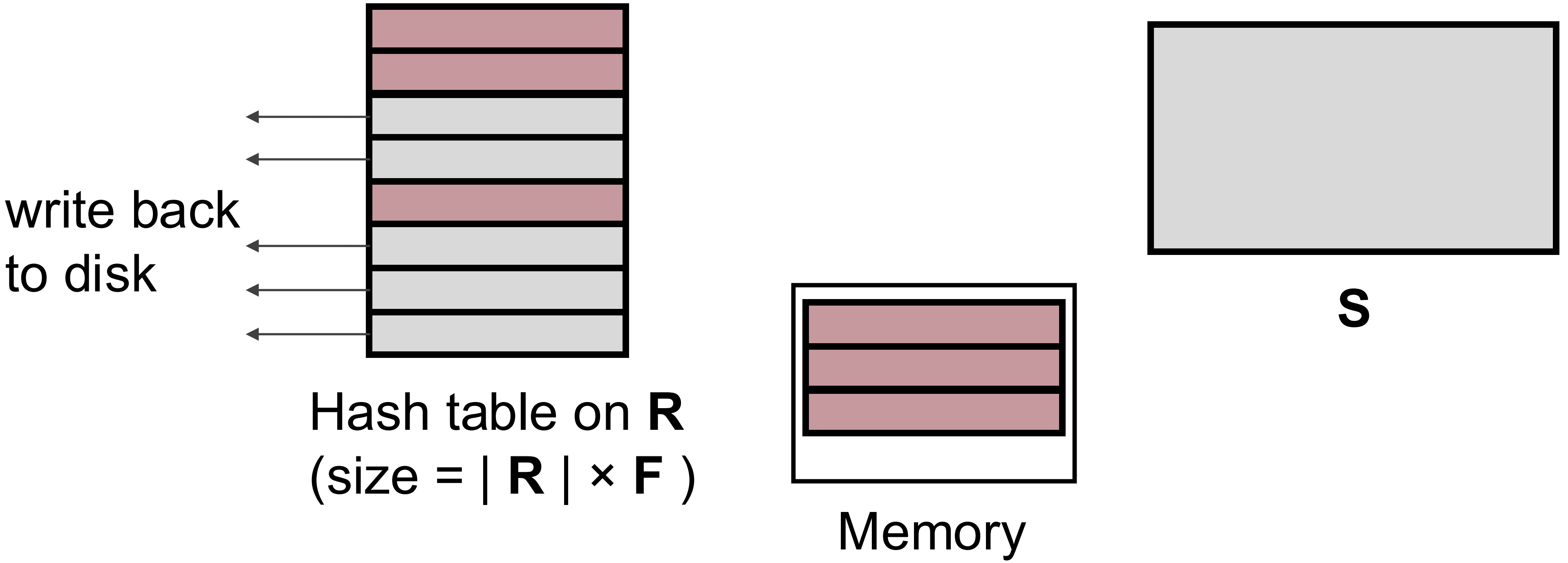
Simple Hash Join

- Build a hash table on **R**



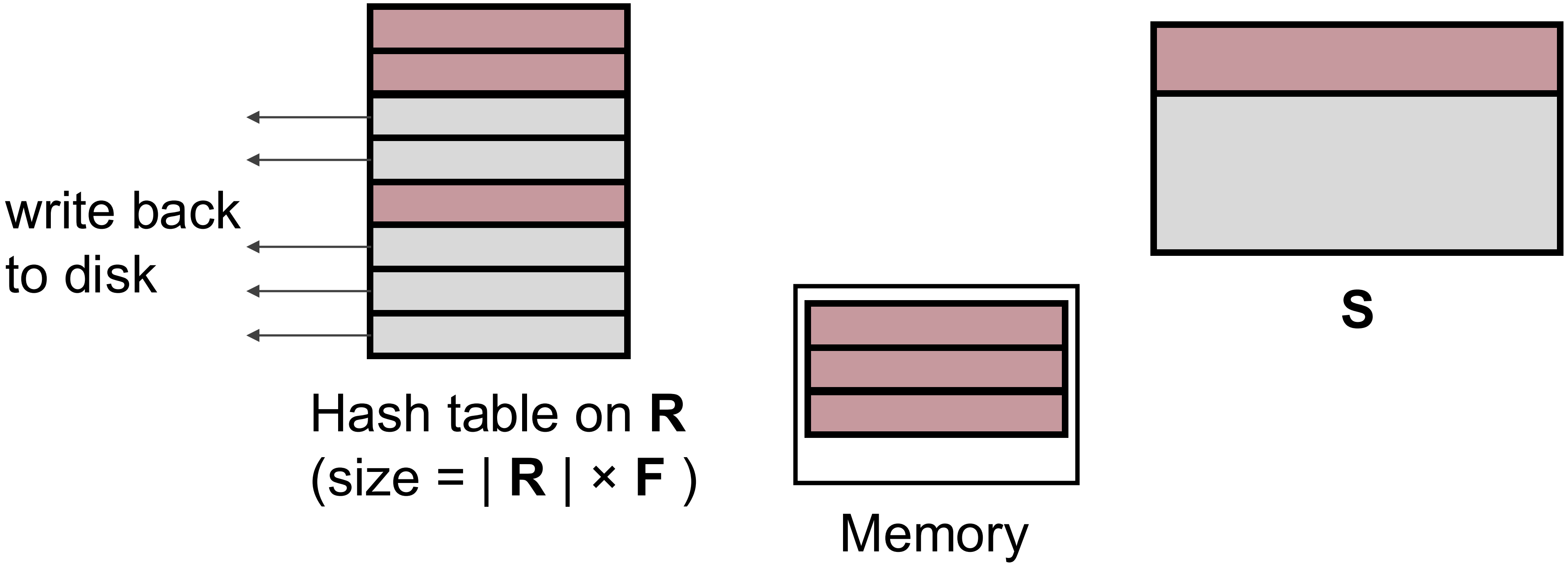
Simple Hash Join – 1st pass

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



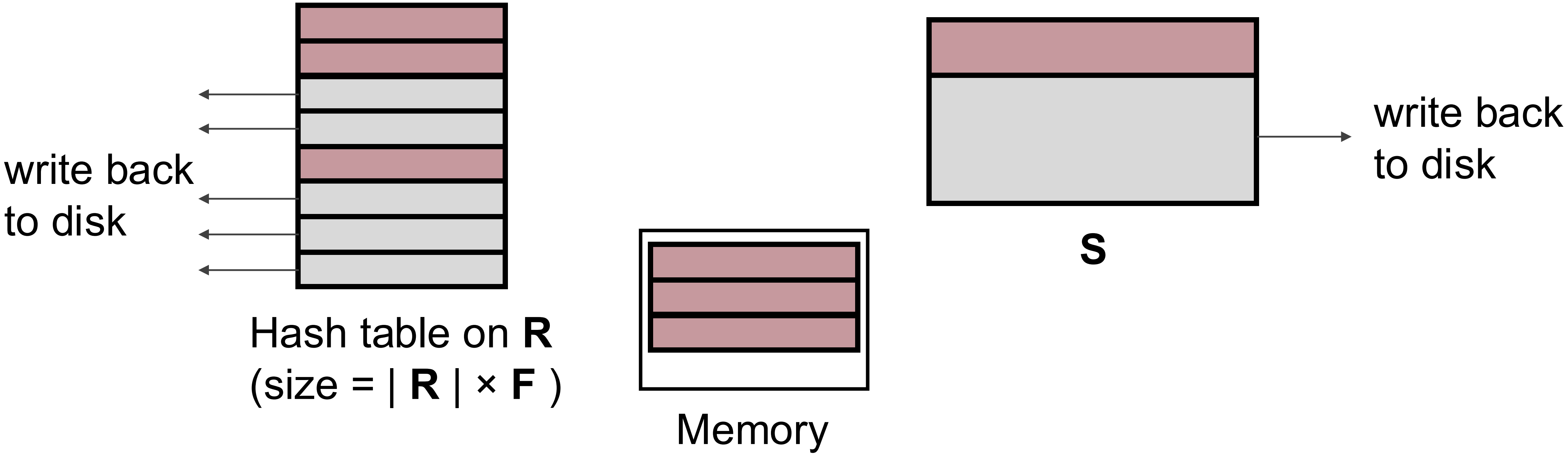
Simple Hash Join – 1st pass

- 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**



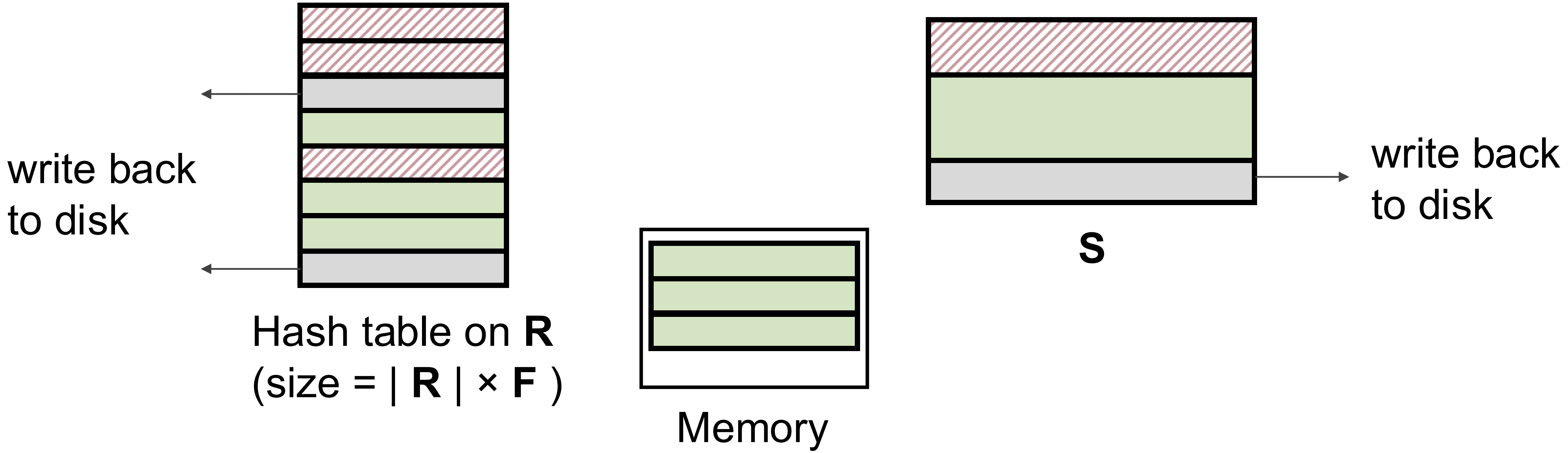
Simple Hash Join – 1^{st} pass

- 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



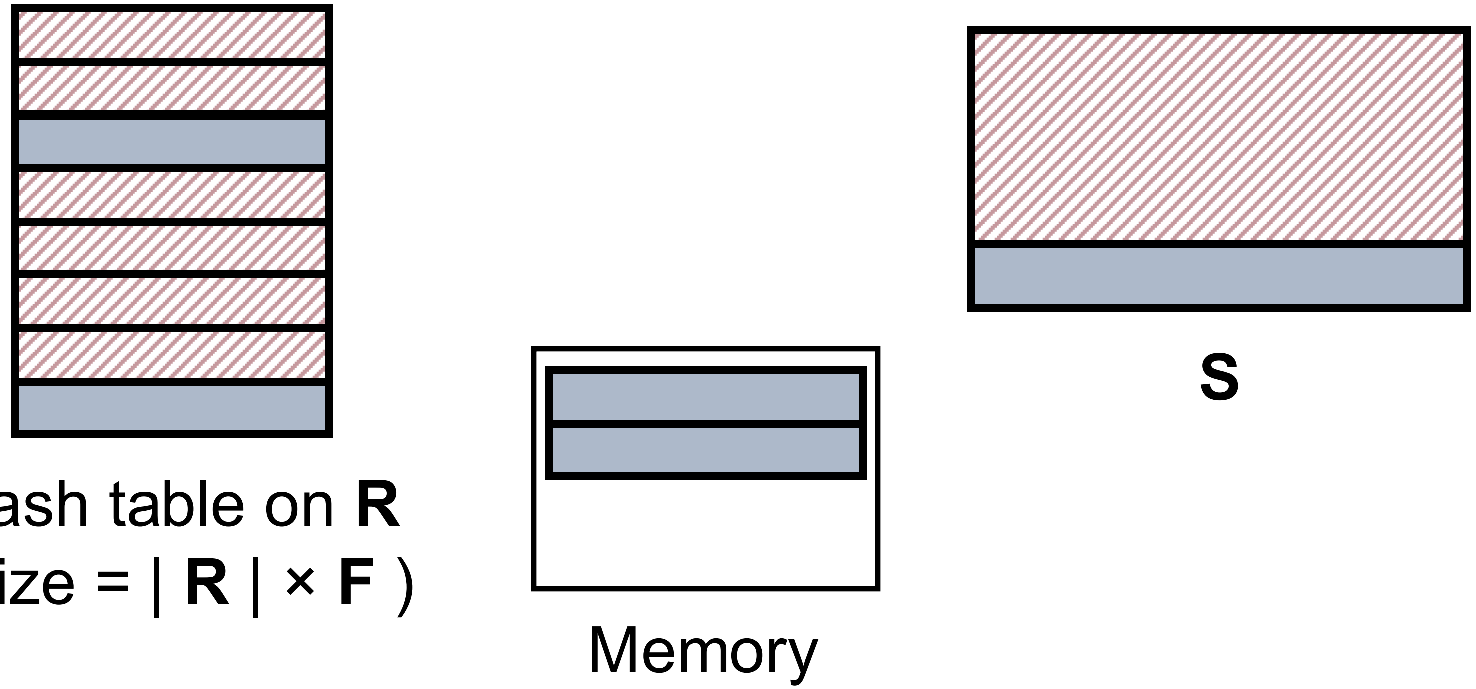
Simple Hash Join – 2nd pass

- 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



Agenda

System architecture and notations

Join algorithms

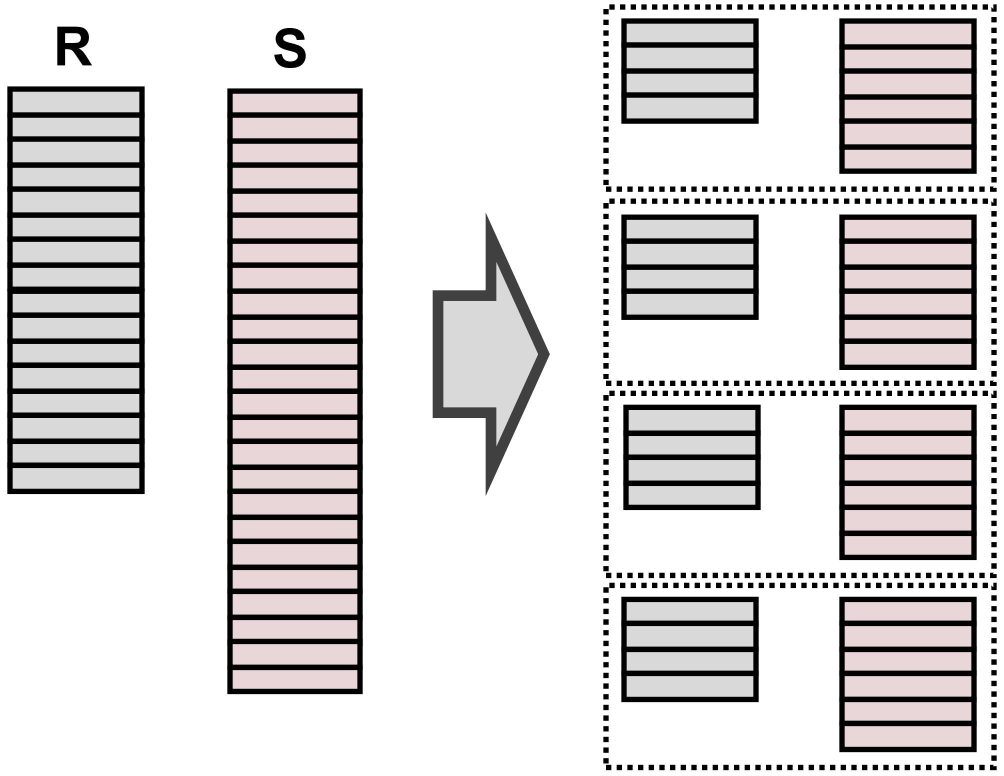
- Sort merge join
- Simple hash join
- **GRACE hash join**
- Hybrid hash join

Partition overflow and additional techniques

GRACE Hash Join

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

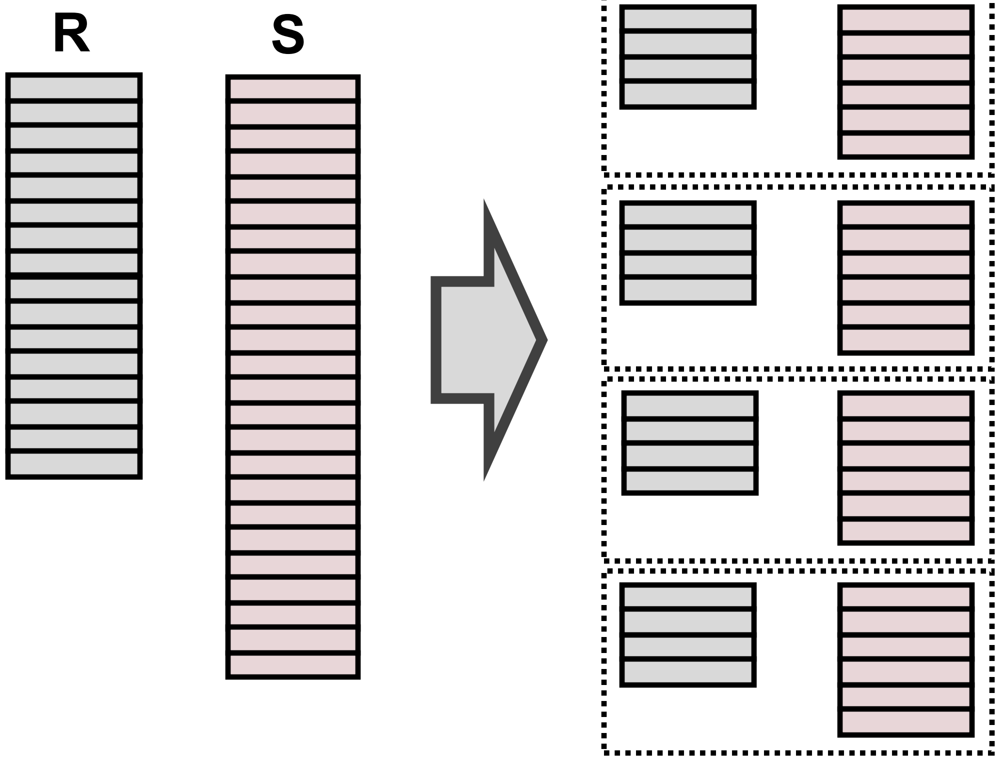
Phase 2: Separately join each pairs of partitions

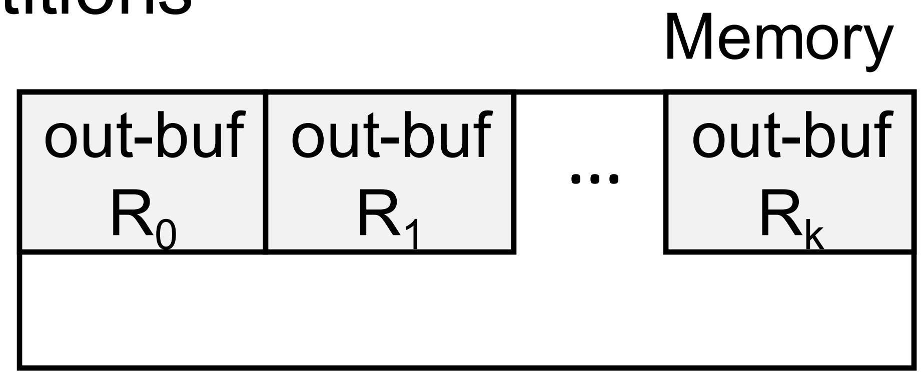


GRACE Hash Join

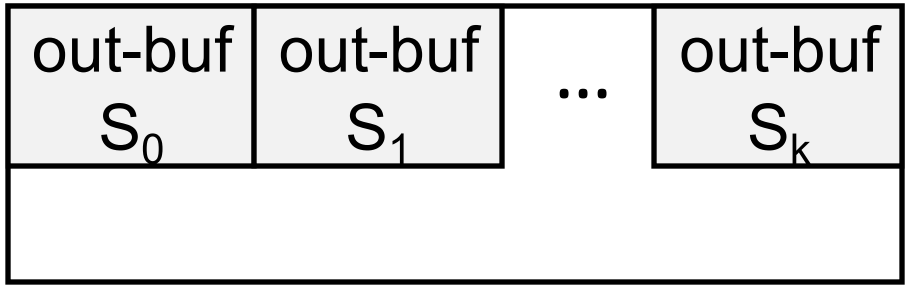
Phase 1: Partition both R and S into pairs of k shards

Phase 2: Separately join each pairs of partitions





Memory layout when Partitioning R

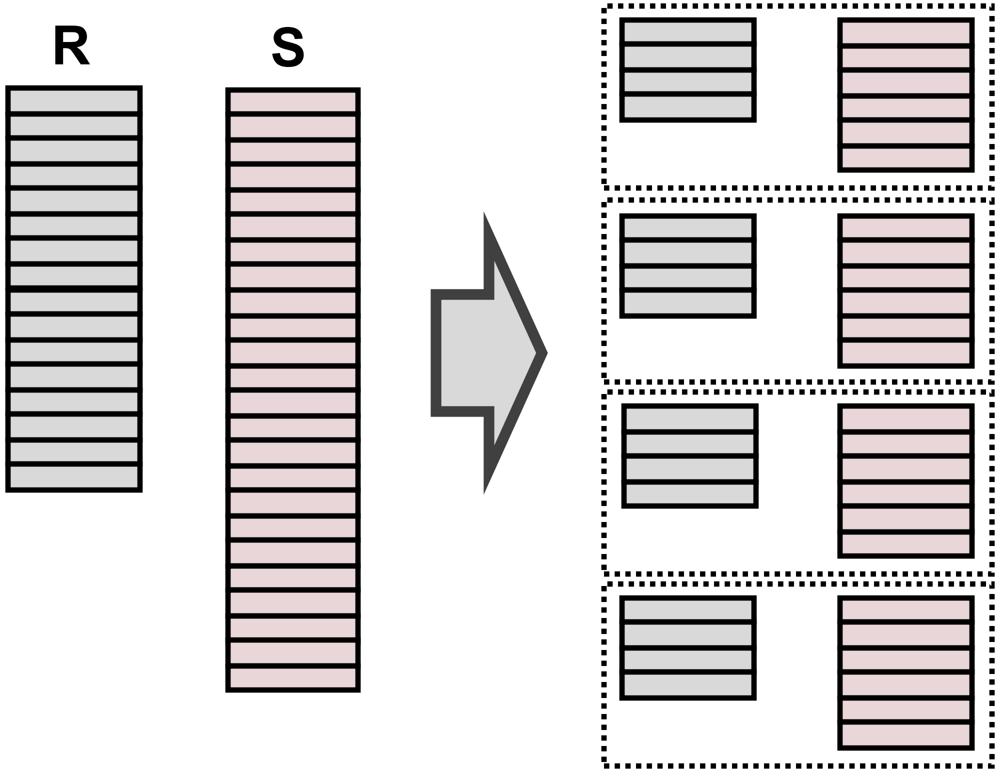


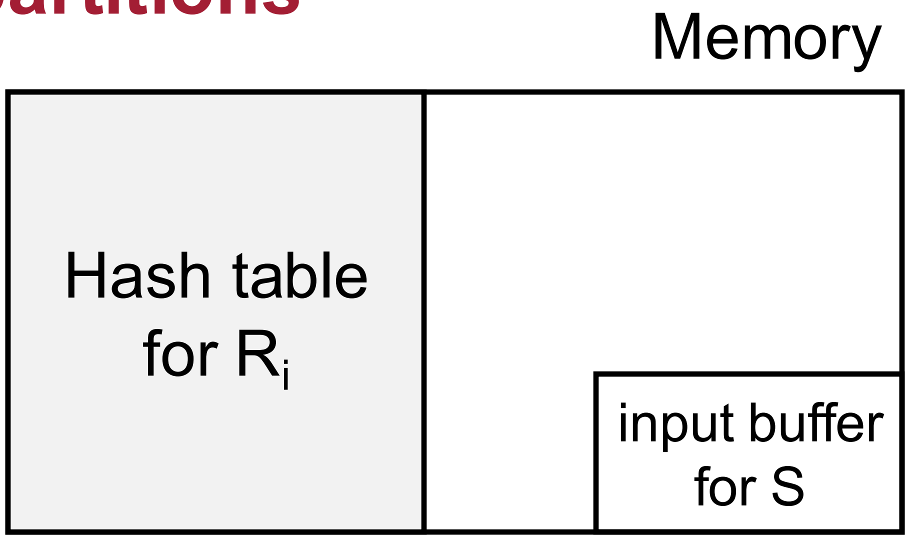
Memory layout when Partitioning S

GRACE Hash Join

Phase 1: Partition both R and S into pairs of k 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 incurs IO linear to table sizes
- Trivial optimization to GRACE: use simple hash join when $| R | \times F < | M |$

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

- Simple hash join incurs significant IO traffic
- GRACE hash join incurs IO linear to table sizes (better)

GRACE vs. Simple Hash Join

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

- Simple hash join incurs no IO traffic (better)
- GRACE hash join incurs IO linear to table sizes
- Trivial optimization to GRACE: use simple hash join when $|R| \times F < |M|$

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

- Simple hash join incurs significant IO traffic
- GRACE hash join incurs IO linear to table sizes (better)

What if $|R| \times F > |M|^2$?

Agenda

System architecture and notations

Join algorithms

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

Partition overflow and additional techniques

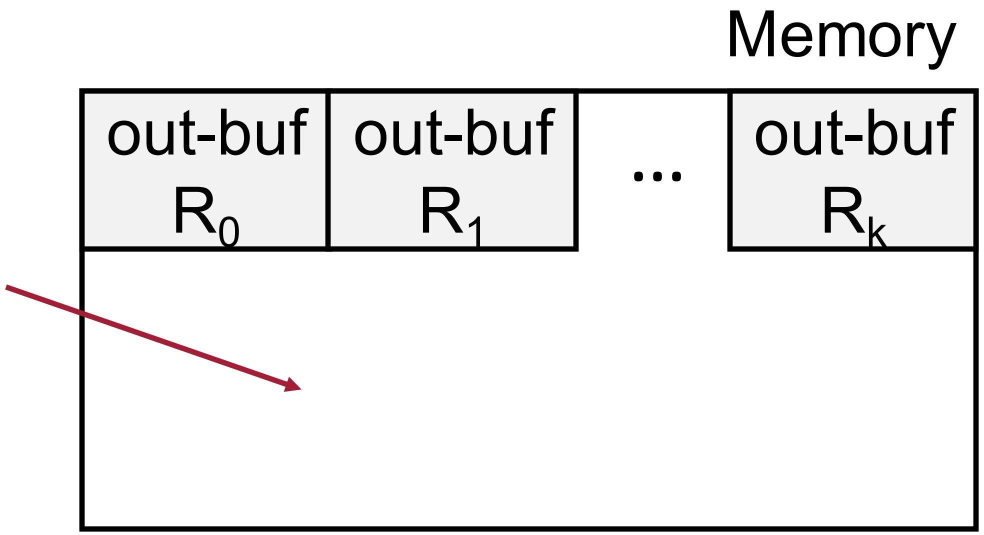
Hybrid Hash Join

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

Hybrid Hash Join

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

Key observation: when | R | is relatively small (e.g., | R | = 2 | M |), significant memory capacity is unused in Phase 1 of GRACE join



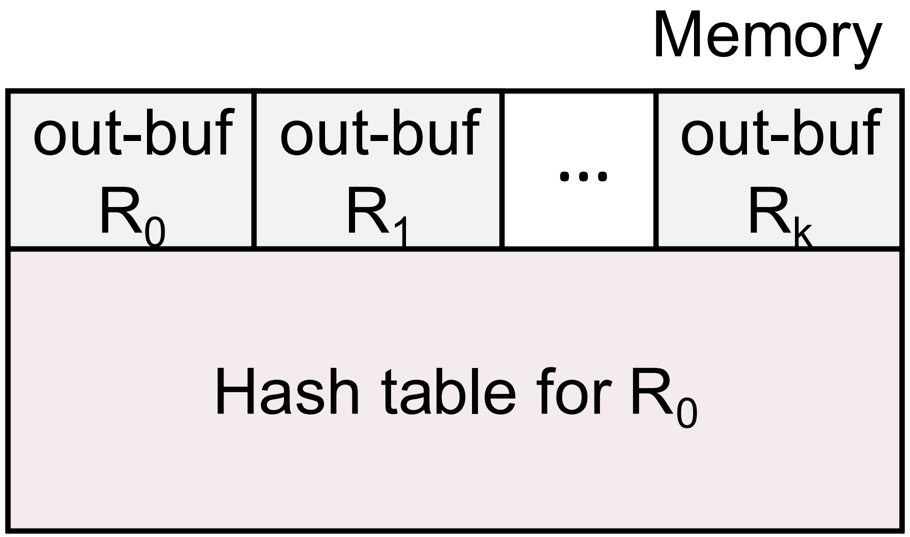
Memory layout in Phase 1 of GRACE hash join

Hybrid Hash Join

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

Key observation: when | R | is relatively small (e.g., | R | = 2 | M |), significant memory capacity is unused in Phase 1 of GRACE join

Key idea: Use the otherwise-unused memory to build hash table for R_0

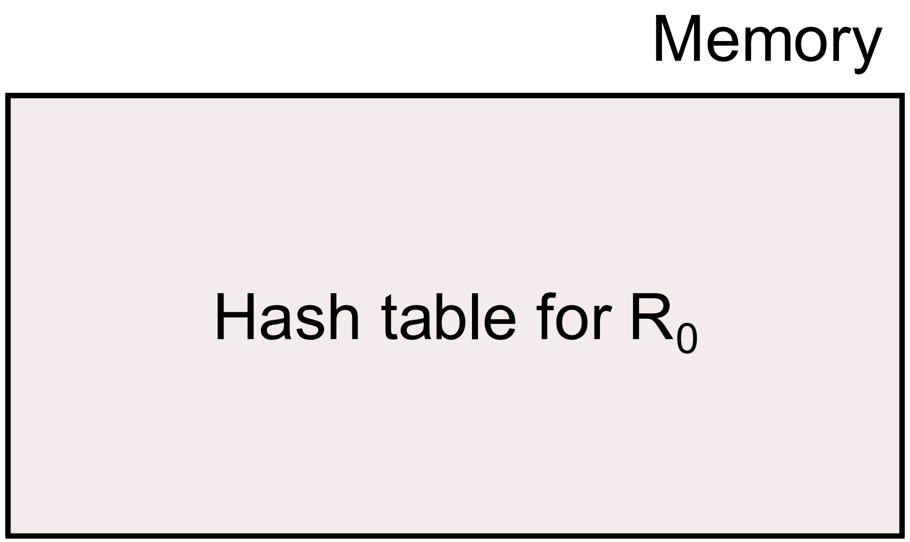


Memory layout in Phase 1 of GRACE hash join

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

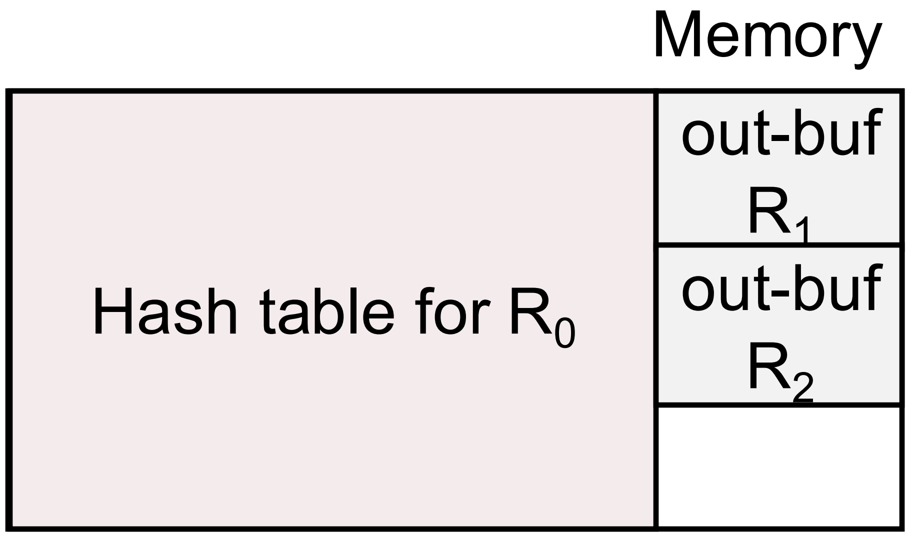
Hybrid Hash Join

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

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

Case 2: $|R| \times F = \alpha |M|$ (α is small)

- R_0 is a significant fraction of R
- R_0 is not written to disk
- Performance is like simple hash join



Memory layout in Phase 1 of hybrid hash join

Hybrid Hash Join

Case 1: | R | × F < | M |

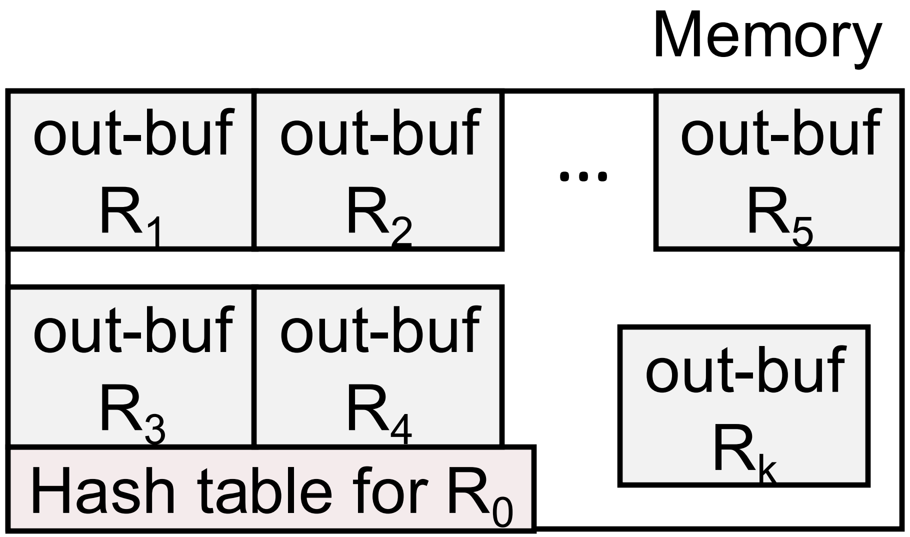
- No need to partition R
- Identical to simple hash join

Case 2: | R | × F = α | M | (α is small)

- R_0 is a significant fraction of R
- R_0 is not written to disk
- Performance is like simple hash join

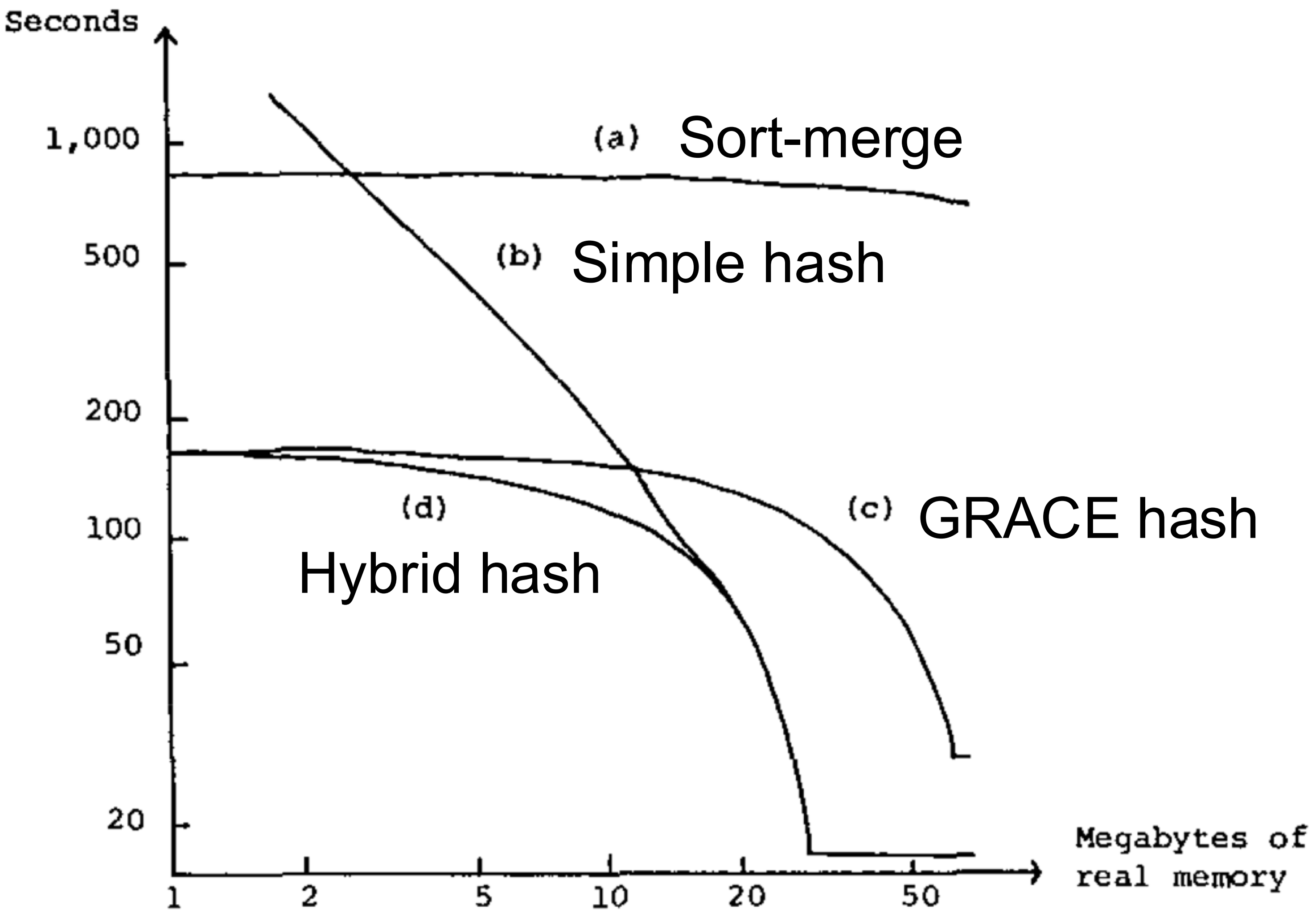
Case 3: | R | × F >> | M |

- R_0 is an insignificant fraction of R
- Performance is like 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

Agenda

System architecture and notations

Join algorithms

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

Partition overflow and additional techniques

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)

- One bit per hash bucket in R
- Set the bit if a tuple in R maps to the bucket
- When scanning S, if a tuple hashes to a bucket where the bit is unset, can discard the tuple immediately

Additional Techniques

Babb array (or bitmap filter)

- One bit per hash bucket in R
- Set the bit if a tuple in R maps to the bucket
- When scanning S, if a tuple hashes to a bucket where the bit is unset, can discard the tuple immediately

Semi-join ($R \ltimes S$)

- $R \ltimes S$ returns tuples in R that have matching rows in S
- Semi-joins can be cheaper than joins, and remove many tuples from R
- Can be added to any join algorithm above

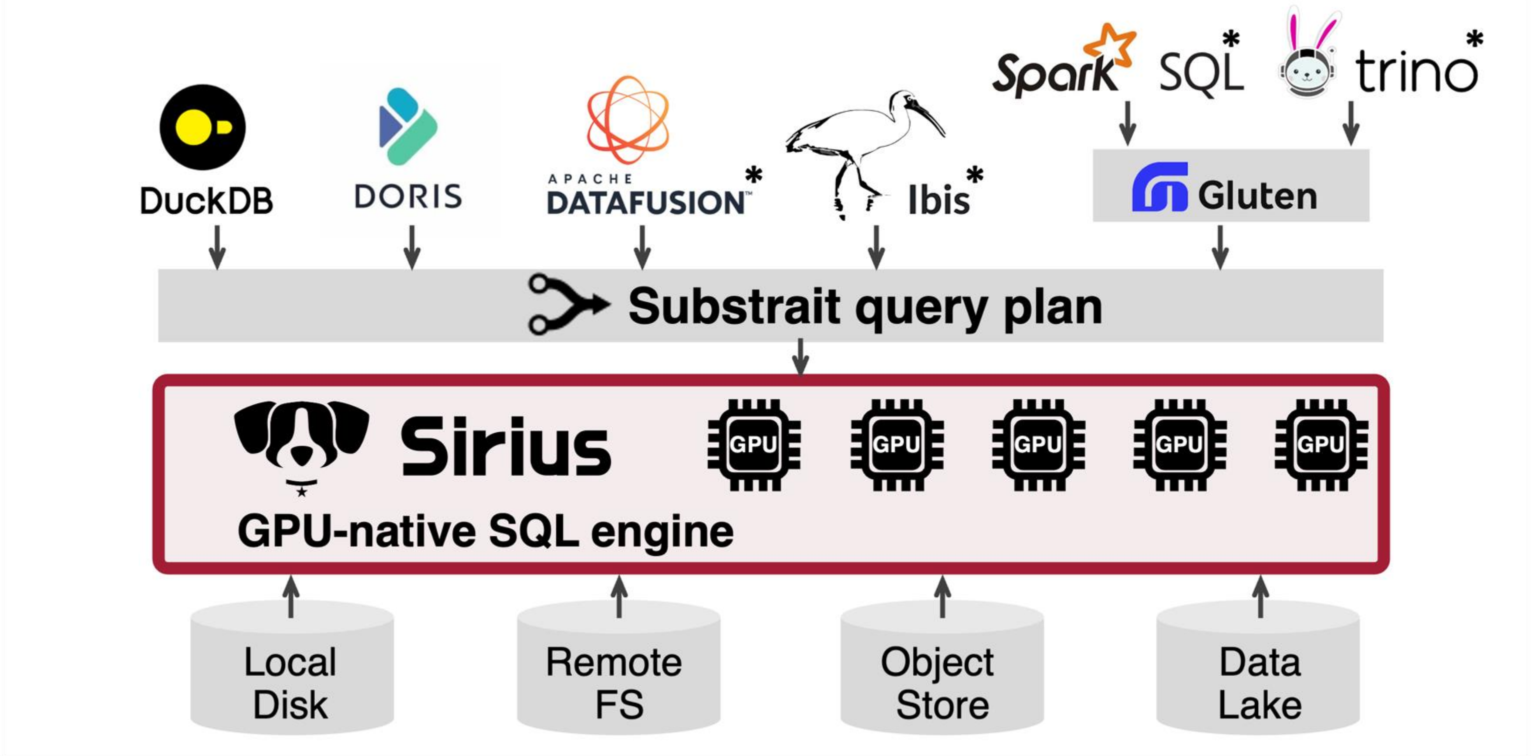
Join – Comments and Q/A

- Will conclusions change on modern hardware?
  - Bigger memory, SSD, HJ vs. SMJ
- Key distribution is super skewed?
- How does a DB choose between SMJ and HJ?
- Join implementations on multicore and distributed system?

Course Project Ideas

Sirius (https://github.com/sirius-db/sirius)

– GPU-native SQL engine



Sirius is **60x** faster than ClickHouse and **8.2x** faster than DuckDB

Ideas Related to Sirius

- Run more benchmarks: JOB, TPC-DS; report and fix bugs
- **Help Sirius become a community DuckDB extension**
- Support reading Parquet (with and without compression)
- Support reading data from S3
- Support more operators (e.g., window functions, intersection, union, etc.)
- Support variable length data types (e.g., array, list, struct, map)
- Optimize string, top-K, **regex (e.g., fallback to DuckDB)** performance
- **Support vector data type in Sirius**
- Support compression for integer, floating point, and string
- **Graph queries on Sirius (e.g., shortest path)**

Before Next Lecture

Submit review for

Yifei Yang, et al., Predicate Transfer: Efficient Pre-Filtering on Multi-Join Queries. CIDR 2024