Intro to Databases (COMP_SCI 339)

09 Join Algorithms

Northwestern University

WINTER 2024

Andrew Crotty

ADMINISTRIVIA

Project #3 is due Sunday 2/18 @ 11:59pm

Exam #2 will be on 2/14 from 3:30-4:50pm

EXAM #2

Who: You

What: Exam #2

Where: Here

When: Wednesday 2/14 from 3:30-4:50pm

What to bring:

→ Pencil or pen with dark-colored ink

→ <u>**One**</u> double-sided 8.5x11" page of handwritten notes

WHY DO WE NEED TO JOIN?

We normalize tables in a relational database to avoid unnecessary repetition of information.

We then use the **join operator** to reconstruct the original tuples without any information loss.

JOIN ALGORITHMS

We will focus on performing binary joins (two tables) using **inner equijoin** algorithms.

→ These algorithms can be tweaked to support other joins.

→ Multi-way joins exist primarily in research literature.

In general, we want the smaller table to always be the left table ("outer table") in the query plan.

→ The optimizer will (try to) figure this out when generating the physical plan.

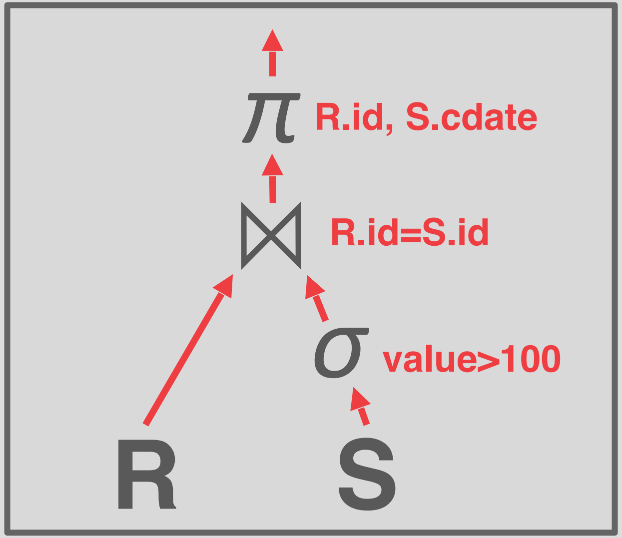
QUERY PLAN

The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.
→ We will discuss the granularity of the data movement later.

The output of the root node is the result of the query.

```
SELECT R.id, S.cdate
FROM R JOIN S
    ON R.id = S.id
WHERE S.value > 100
```



JOIN OPERATORS

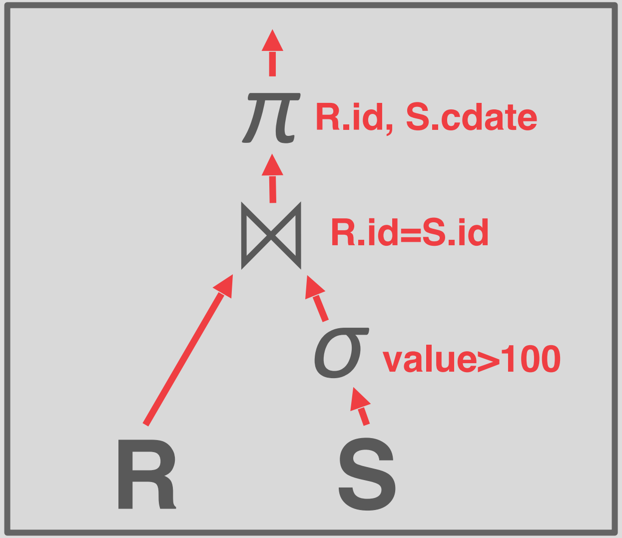
Decision #1: Output

→ What data does the join operator emit to its parent operator in the query plan tree?

Decision #2: Cost Analysis Criteria

→ How do we determine whether one join algorithm is better than another?

```
SELECT R.id, S.cdate
FROM R JOIN S
   ON R.id = S.id
WHERE S.value > 100
```



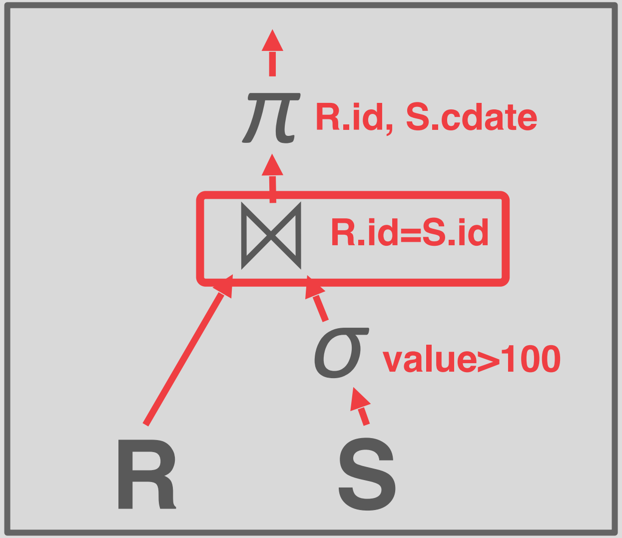
OPERATOR OUTPUT

For tuple **r∈R** and tuple **s∈S** that match on join attributes, concatenate **r** and **s** together into a new tuple.

Output contents can vary:

→ Depends on processing model
→ Depends on storage model
→ Depends on data requirements in query

```
SELECT R.id, S.cdate
FROM R JOIN S
  ON R.id = S.id
WHERE S.value > 100
```



OPERATOR OUTPUT: DATA

Early Materialization:

→ Copy the values for the attributes in outer and inner tuples into a new output tuple.

Subsequent operators in the query plan never need to go back to the base tables to get more data.

```
SELECT R.id, S.cdate
FROM R JOIN S
    ON R.id = S.id
WHERE S.value > 100
```

R(id,name)

id	name
123	abc

⋈

S(id,value,cdate)

id	value	cdate
123	1000	10/5/2022
123	2000	10/5/2022

R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/5/2022
123	abc	123	2000	10/5/2022

OPERATOR OUTPUT: RECORD IDS

Late Materialization:

→ Only copy the joins keys along with the Record IDs of the matching tuples.

Ideal for column stores because the DBMS does not copy data that is not needed for the query.

```
SELECT R.id, S.cdate
FROM R JOIN S
    ON R.id = S.id
WHERE S.value > 100
```

R(id,name)

id	name
123	abc

⋈

S(id,value,cdate)

id	value	cdate
123	1000	10/5/2022
123	2000	10/5/2022

R.id	R.RID	S.id	S.RID
123	R.###	123	S.###
123	R.###	123	S.###

COST ANALYSIS CRITERIA

```
SELECT R.id, S.cdate
FROM R JOIN S
   ON R.id = S.id
WHERE S.value > 100
```

Assume:

→ ***M*** pages in table **R**, ***m*** tuples in **R**

→ ***N*** pages in table **S**, ***n*** tuples in **S**

Cost Metric: # of IOs to compute join

We will ignore output costs since that depends on the data and we cannot compute that yet.

JOIN VS CROSS-PRODUCT

R⋈S is the most common operation and thus must be carefully optimized.

R×S followed by a selection is inefficient because the cross-product is large.

There are many algorithms for reducing join cost, but no algorithm works well in all scenarios.

JOIN ALGORITHMS

Nested Loop Join
- → Simple
- → Block
- → Index

Sort-Merge Join

Hash Join

NESTED LOOP JOIN

```
foreach tuple r ∈ R:
  foreach tuple s ∈ S:
    emit, if r and s match
```

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

NESTED LOOP JOIN

```
foreach tuple r ∈ R:        ← Outer
  foreach tuple s ∈ S:      ← Inner
    emit, if r and s match
```

R(id,name)

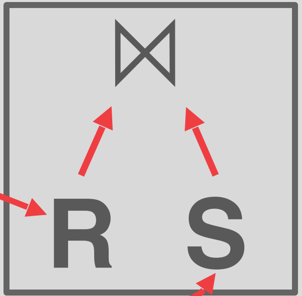
id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

NESTED LOOP JOIN

foreach tuple $r \in R$: ← Outer
** foreach tuple $s \in S$:** ← Inner
**   emit, if r and s match**



R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

SIMPLE NESTED LOOP JOIN

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

SIMPLE NESTED LOOP JOIN

Is this a good algorithm?
→ For every tuple in **R**, it scans **S** once

Cost: M + (m · N)

M pages
m tuples

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

N pages
n tuples

SIMPLE NESTED LOOP JOIN

Example database:

→ **Table R**: $M = 1000$, $m = 100{,}000$

→ **Table S**: $N = 500$, $n = 40{,}000$

SIMPLE NESTED LOOP JOIN

Example database:

→ **Table R**: M = 1000, m = 100,000

→ **Table S**: N = 500, n = 40,000

Cost Analysis:

→ $M + (m \cdot N)$ = 1000 + (100000 · 500) = **50,001,000 IOs**

→ At 0.1 ms/IO, Total time ≈ 1.3 hours

SIMPLE NESTED LOOP JOIN

Example database:
→ **Table R**: M = 1000, m = 100,000
→ **Table S**: N = 500, n = 40,000

4 KB pages → 6 MB

Cost Analysis:
→ $M + (m \cdot N)$ = 1000 + (100000 · 500) = **50,001,000 IOs**
→ At 0.1 ms/IO, Total time ≈ 1.3 hours

What if smaller table (**S**) is used as the outer table?
→ $N + (n \cdot M)$ = 500 + (40000 · 1000) = **40,000,500 IOs**
→ At 0.1 ms/IO, Total time ≈ 1.1 hours

BLOCK NESTED LOOP JOIN

```
foreach block B_R ∈ R:
  foreach block B_S ∈ S:
    foreach tuple r ∈ B_R:
      foreach tuple s ∈ B_s:
        emit, if r and s match
```

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

N pages
n tuples

BLOCK NESTED LOOP JOIN

This algorithm performs fewer disk accesses.
→ For every block in **R**, it scans **S** once.

Cost: $M + (M \cdot N)$

M pages
m tuples

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

N pages
n tuples

BLOCK NESTED LOOP JOIN

The smaller table should be the outer table.

We determine size based on the number of pages, not the number of tuples.

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

N pages
n tuples

BLOCK NESTED LOOP JOIN

Example database:

→ **Table R**: $M = 1000$, $m = 100{,}000$

→ **Table S**: $N = 500$, $n = 40{,}000$

Cost Analysis:

→ $M + (M \cdot N) = 1000 + (1000 \cdot 500) =$ **501,000 IOs**

→ At 0.1 ms/IO, Total time ≈ 50 seconds

BLOCK NESTED LOOP JOIN

What if we have ***B*** buffers available?

→ Use ***B*-2** buffers for scanning the outer table.

→ Use one buffer for the inner table, one buffer for storing output.

M pages
m tuples

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

N pages
n tuples

BLOCK NESTED LOOP JOIN

```
foreach B-2 pages pR ∈ R:
  foreach page pS ∈ S:
    foreach tuple r ∈ B-2 pages:
      foreach tuple s ∈ ps:
        emit, if r and s match
```

M pages
m tuples

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

N pages
n tuples

BLOCK NESTED LOOP JOIN

This algorithm uses ***B*-2** buffers for scanning **R**.

Cost: $M + (\lceil M / (B\text{-}2) \rceil \cdot N)$

BLOCK NESTED LOOP JOIN

This algorithm uses ***B*-2** buffers for scanning **R**.

Cost: $M + (\lceil M / (B\text{-}2) \rceil \cdot N)$

What if the outer relation completely fits in memory ($B > M+2$)?

BLOCK NESTED LOOP JOIN

This algorithm uses ***B*-2** buffers for scanning **R**.

Cost: $M + (\lceil M / (B\text{-}2) \rceil \cdot N)$

What if the outer relation completely fits in memory (**$B > M+2$**)?

→ **Cost: $M + N$** = 1000 + 500 = **1500 IOs**

→ At 0.1ms/IO, Total time ≈ 0.15 seconds

NESTED LOOP JOIN

Why is the basic nested loop join so bad?

→ For each tuple in the outer table, we must do a sequential scan to check for a match in the inner table.

NESTED LOOP JOIN

Why is the basic nested loop join so bad?

→ For each tuple in the outer table, we must do a sequential scan to check for a match in the inner table.

We can avoid sequential scans by using an index to find inner table matches.

→ Use an existing index for the join.

INDEX NESTED LOOP JOIN

```
foreach tuple r ∈ R:
  foreach tuple s ∈ Index(r_i = s_j):
    emit, if r and s match
```

Index(S.id)

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

N pages
n tuples

INDEX NESTED LOOP JOIN

Assume the cost of each index probe is some constant *C* per tuple.

Cost: $M + (m \cdot C)$

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

Index(S.id)

N pages
n tuples

NESTED LOOP JOIN: SUMMARY

Key Takeaways

→ Pick the smaller table as the outer table.
→ Buffer as much of the outer table in memory as possible.
→ Loop over the inner table (or use an index).

Algorithms

→ Simple
→ Block
→ Index

SORT-MERGE JOIN

Phase #1: Sort

→ Sort both tables on the join key(s).
→ We can use the external merge sort algorithm that we talked about last class.

Phase #2: Merge

→ Step through the two sorted tables with cursors and emit matching tuples.
→ May need to backtrack depending on the join type.

SORT-MERGE JOIN

```
sort R,S on join keys
cursor_R ← R_sorted, cursor_S ← S_sorted
while cursor_R and cursor_S:
  if cursor_R > cursor_S:
    increment cursor_S
  if cursor_R < cursor_S:
    increment cursor_R
  elif cursor_R and cursor_S match:
    emit
    increment cursor_S
```

SORT-MERGE JOIN

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
200	GZA
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

SORT-MERGE JOIN

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
200	GZA
400	Raekwon

Sort!

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
500	7777	10/5/2022
400	6666	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022

Sort!

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

Sort!

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

Sort!

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
  WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
  WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022
400	Raekwon	200	6666	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022
400	Raekwon	200	6666	10/5/2022
500	RZA	500	7777	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022
400	Raekwon	200	6666	10/5/2022
500	RZA	500	7777	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022
400	Raekwon	200	6666	10/5/2022
500	RZA	500	7777	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022
400	Raekwon	200	6666	10/5/2022
500	RZA	500	7777	10/5/2022

SORT-MERGE JOIN

R(id,name)

id	name
100	Andy
200	GZA
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/5/2022
100	9999	10/5/2022
200	8888	10/5/2022
400	6666	10/5/2022
500	7777	10/5/2022

```
SELECT R.id, S.cdate
  FROM R JOIN S
   ON R.id = S.id
 WHERE S.value > 100
```

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/5/2022
100	Andy	100	9999	10/5/2022
200	GZA	200	8888	10/5/2022
200	GZA	200	8888	10/5/2022
400	Raekwon	200	6666	10/5/2022
500	RZA	500	7777	10/5/2022

SORT-MERGE JOIN

Sort Cost (R): $2M \cdot (1 + \lceil \log_{B-1} \lceil M / B \rceil \rceil)$

Sort Cost (S): $2N \cdot (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$

Merge Cost: $(M + N)$

Total Cost: Sort + Merge

SORT-MERGE JOIN

Example database:
→ **Table R**: M = 1000, m = 100,000
→ **Table S**: N = 500, n = 40,000

With B=100 buffer pages, both **R** and **S** can be sorted in two passes:
→ Sort Cost (**R**) = $2000 \cdot (1 + \lceil \log_{99} 1000 / 100 \rceil)$ = **4000 IOs**
→ Sort Cost (**S**) = $1000 \cdot (1 + \lceil \log_{99} 500 / 100 \rceil)$ = **2000 IOs**
→ Merge Cost = (1000 + 500) = **1500 IOs**
→ Total Cost = 4000 + 2000 + 1500 = **7500 IOs**
→ At 0.1 ms/IO, Total time ≈ 0.75 seconds

SORT-MERGE JOIN

The worst case for the merging phase is when the join attribute of all the tuples in both relations contains the same value.

Cost: (M · N) + (sort cost)

WHEN IS SORT-MERGE JOIN USEFUL?

One or both tables are already sorted on join key.

Output must be sorted on join key.

The input relations may be sorted either by an explicit sort operator, or by scanning the relation using an index on the join key.

HASH JOIN

If tuple **$r \in R$** and a tuple **$s \in S$** satisfy the join condition, then they have the same value for the join attributes.

If that value is hashed to some partition **i**, the **R** tuple must be in **r_i** and the **S** tuple in **s_i**.

Therefore, **R** tuples in **r_i** need only to be compared with **S** tuples in **s_i**.

BASIC HASH JOIN ALGORITHM

Phase #1: Build

→ Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

→ We can use any hash table that we discussed before but in practice linear probing works the best.

Phase #2: Probe

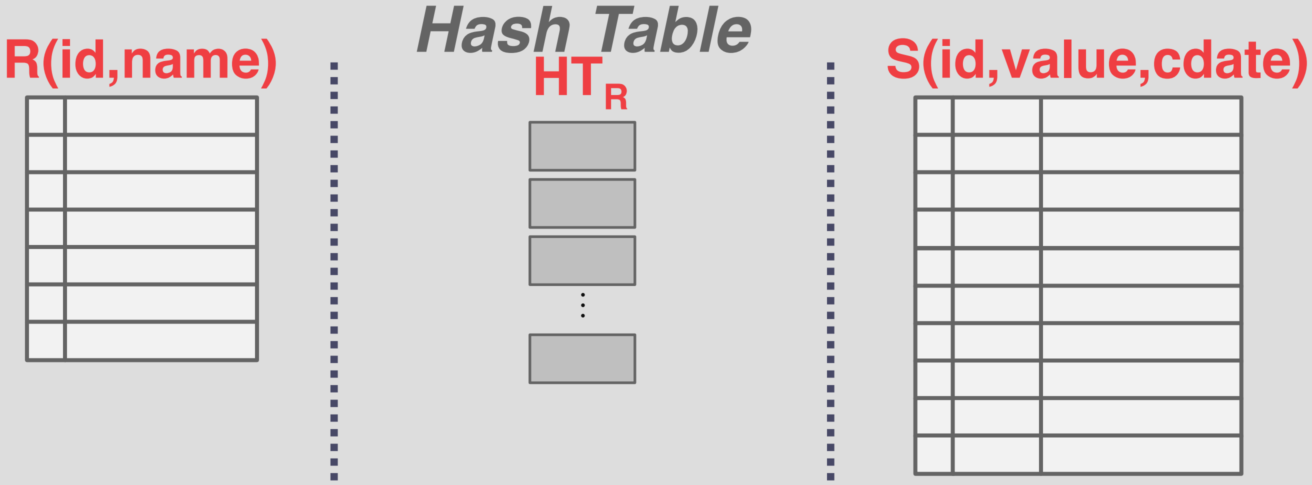
→ Scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

BASIC HASH JOIN ALGORITHM

build hash table HT_R for R
foreach tuple $s \in S$
  output, if $h_1(s) \in HT_R$

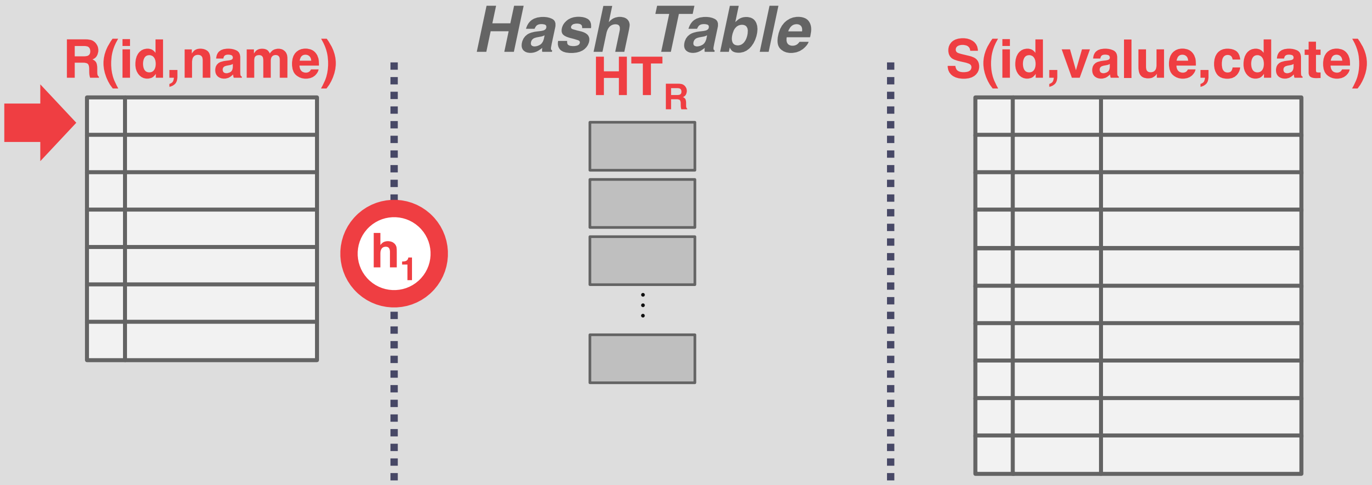
BASIC HASH JOIN ALGORITHM

build hash table HT_R for R
foreach tuple $s \in S$
output, if $h_1(s) \in HT_R$



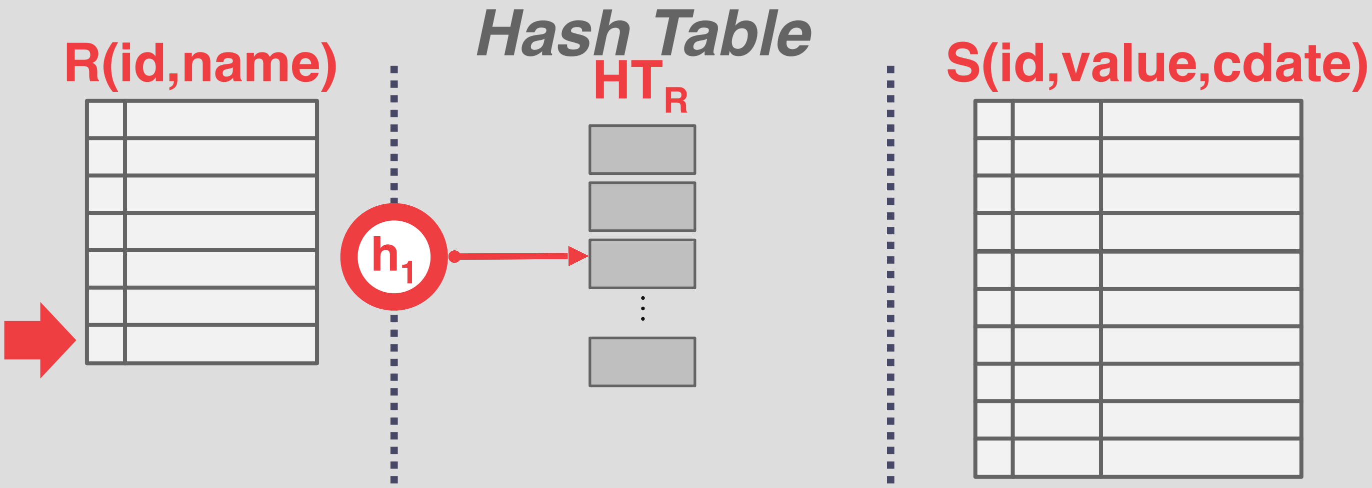
BASIC HASH JOIN ALGORITHM

build hash table HT_R for R
foreach tuple $s \in S$
output, if $h_1(s) \in HT_R$



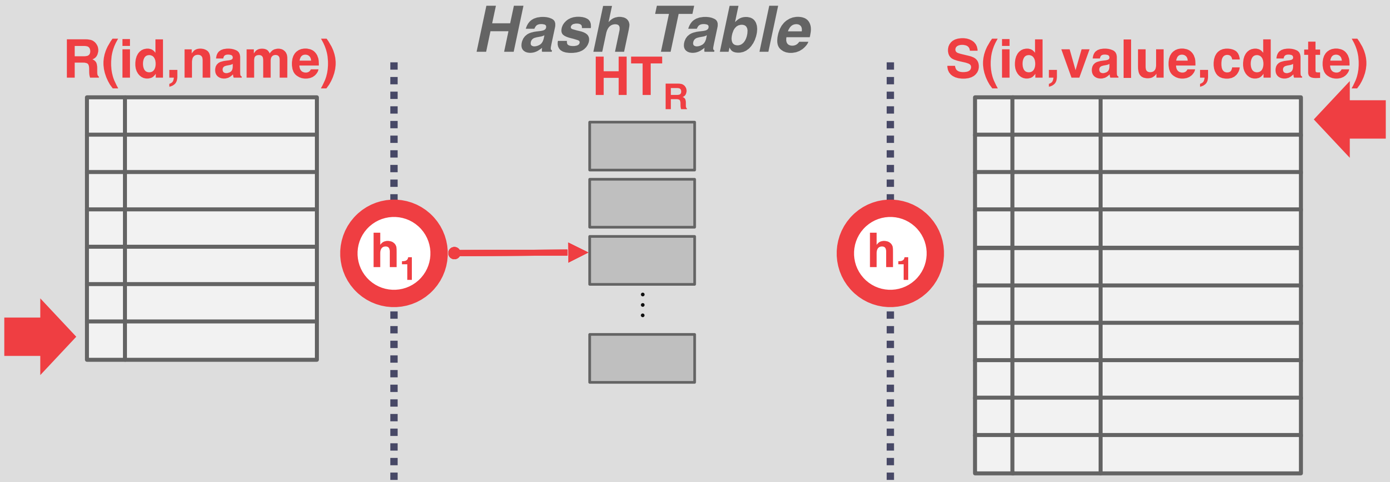
BASIC HASH JOIN ALGORITHM

build hash table HT_R for R
foreach tuple $s \in S$
output, if $h_1(s) \in HT_R$



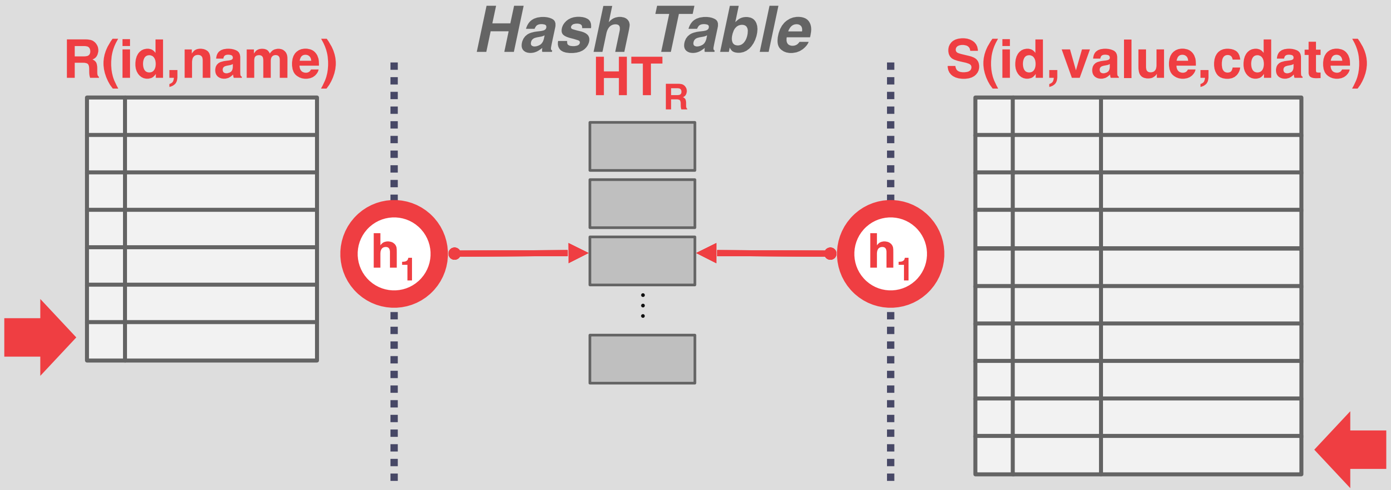
BASIC HASH JOIN ALGORITHM

build hash table HT_R for R
foreach tuple $s \in S$
output, if $h_1(s) \in HT_R$



BASIC HASH JOIN ALGORITHM

build hash table HT_R for R
foreach tuple $s \in S$
output, if $h_1(s) \in HT_R$



HASH TABLE CONTENTS

Key: The attribute(s) that the query is joining the tables on.

→ We always need the original key to verify that we have a correct match in case of hash collisions.

Value: Varies per implementation.

→ Depends on what the operators above the join in the query plan expect as its input.

→ Early vs. Late Materialization

OPTIMIZATION: PROBE FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

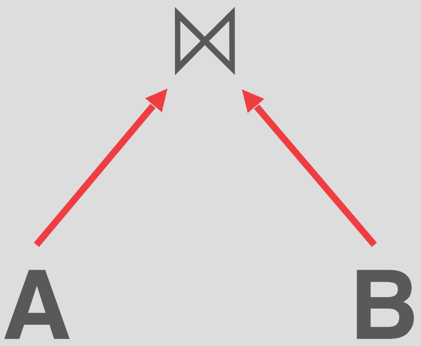
→ Sometimes called ***sideways information passing.***

OPTIMIZATION: PROBE FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***

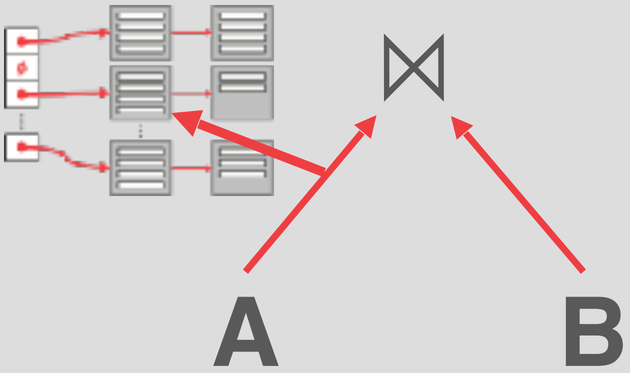


OPTIMIZATION: PROBE FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***

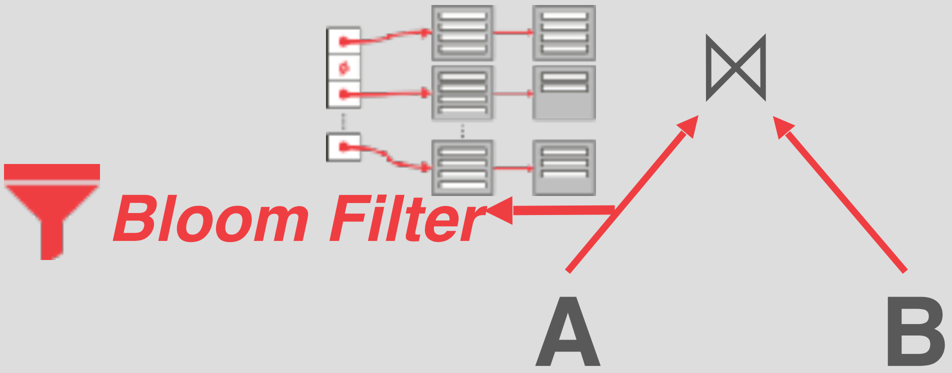


OPTIMIZATION: PROBE FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***

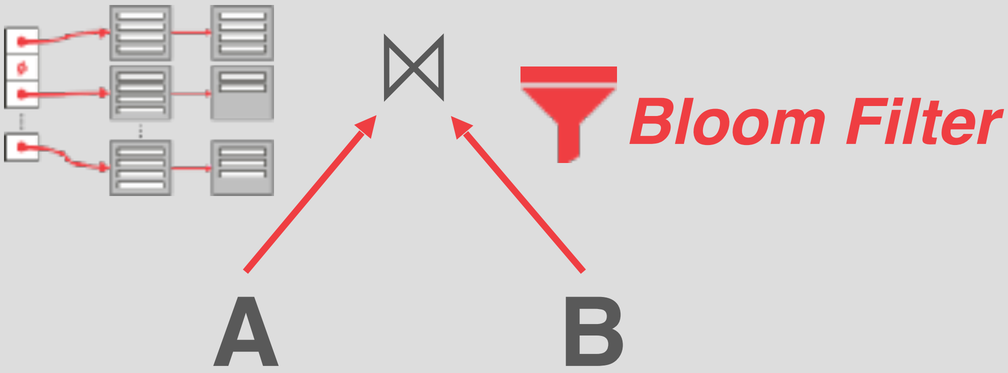


OPTIMIZATION: PROBE FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***

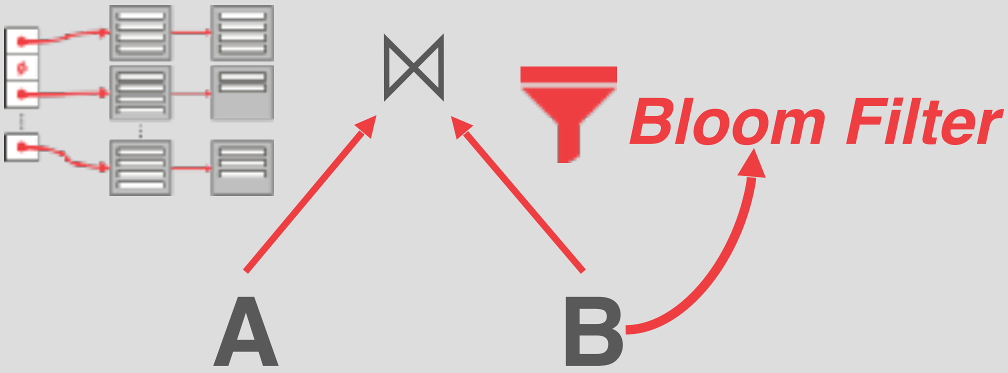


OPTIMIZATION: PROBE FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***

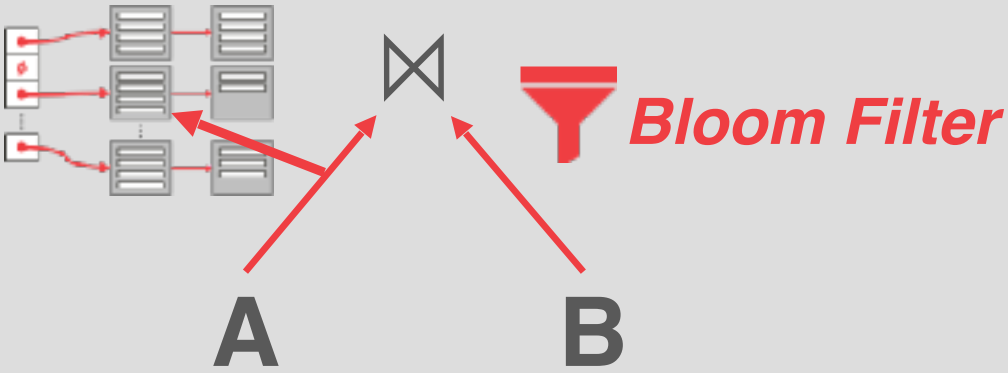


OPTIMIZATION: PROBE FILTER

Create a Bloom Filter during the build phase when the key is likely to not exist in the hash table.

→ Threads check the filter before probing the hash table. This will be faster since the filter will fit in CPU caches.

→ Sometimes called ***sideways information passing.***



BLOOM FILTERS

Probabilistic data structure (bitmap) that answers set membership queries.

→ False negatives will never occur.
→ False positives can sometimes occur.
→ See Bloom Filter Calculator.

Insert(x):

→ Use *k* hash functions to set bits in the filter to 1.

Lookup(x):

→ Check whether the bits are 1 for each hash function.

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Insert('RZA')

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

Insert('RZA')

$hash_1('RZA') = 2222 \% 8 = 6$

$hash_2('RZA') = 4444 \% 8 = 4$

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0

Insert('RZA')

$hash_1$('RZA') = 2222 % 8 = 6

$hash_2$('RZA') = 4444 % 8 = 4

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	0	0	0	1	0	1	0

Insert('RZA')

Insert('GZA')

$hash_1$('GZA') = 5555 % 8 = 3

$hash_2$('GZA') = 7777 % 8 = 1

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('GZA') = 5555 % 8 = 3

$hash_2$('GZA') = 7777 % 8 = 1

Insert('RZA')

Insert('GZA')

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

Insert('RZA')

Insert('GZA')

Lookup(RZA')

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon')

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon')

$hash_1$('Raekwon') = 3333 % 8 = 5

$hash_2$('Raekwon') = 8899 % 8 = 3

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('Raekwon') = 3333 % 8 = 5

$hash_2$('Raekwon') = 8899 % 8 = 3

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon')

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('Raekwon') = 3333 % 8 = 5

$hash_2$('Raekwon') = 8899 % 8 = 3

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon') → ***FALSE***

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon') → ***FALSE***

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1$('ODB') = 6699 % 8 = 3

$hash_2$('ODB') = 9966 % 8 = 6

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon') → ***FALSE***

Lookup('ODB')

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1(\text{'ODB'}) = 6699 \mathbin{\%} 8 = 3$

$hash_2(\text{'ODB'}) = 9966 \mathbin{\%} 8 = 6$

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon') → ***FALSE***

Lookup('ODB')

BLOOM FILTERS

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1(\text{'ODB'}) = 6699 \% 8 = 3$

$hash_2(\text{'ODB'}) = 9966 \% 8 = 6$

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon') → ***FALSE***

Lookup('ODB') → ***TRUE***

HASH JOIN

What happens if we do not have enough memory to fit the entire hash table?

We do not want to let the buffer pool manager swap out the hash table pages at random.

PARTITIONED HASH JOIN

Hash join when tables do not fit in memory.

→ **Build Phase:** Hash both tables on the join attribute into partitions.

→ **Probe Phase:** Compares tuples in corresponding partitions for each table.

Sometimes called **GRACE Hash Join**.

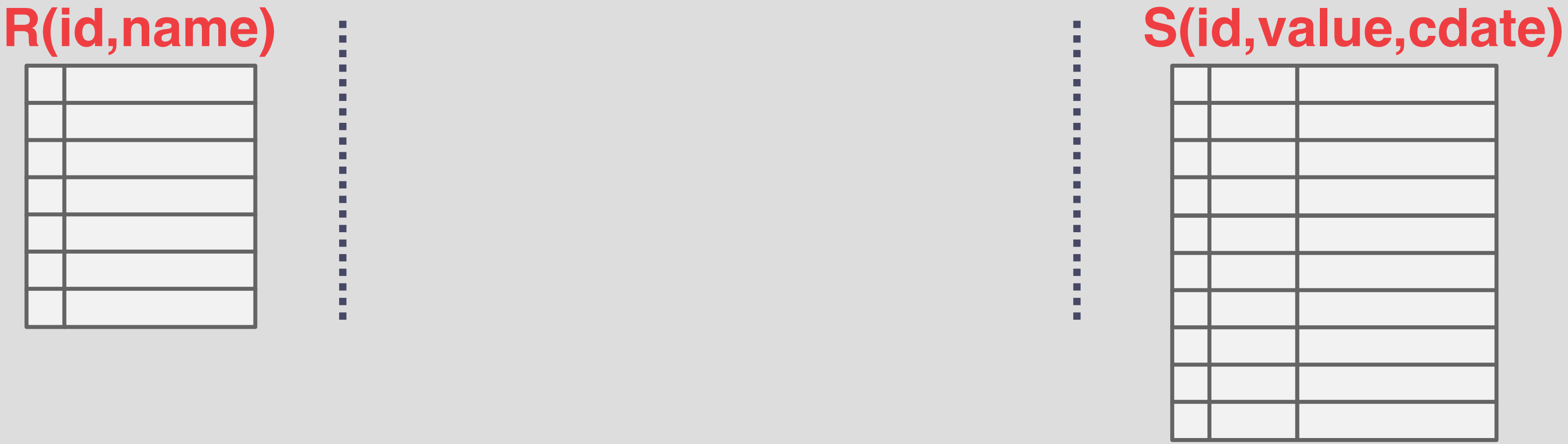
→ Named after the GRACE database machine from Japan in the 1980s.

GRACE
University of Tokyo

PARTITIONED HASH JOIN

Hash **R** into (0, 1, ..., *max*) buckets.

Hash **S** into the same # of buckets with the same hash function.



PARTITIONED HASH JOIN

Hash **R** into (0, 1, ..., *max*) buckets.

Hash **S** into the same # of buckets with the same hash function.

PARTITIONED HASH JOIN

Hash **R** into (0, 1, ..., *max*) buckets.

Hash **S** into the same # of buckets with the same hash function.

PARTITIONED HASH JOIN

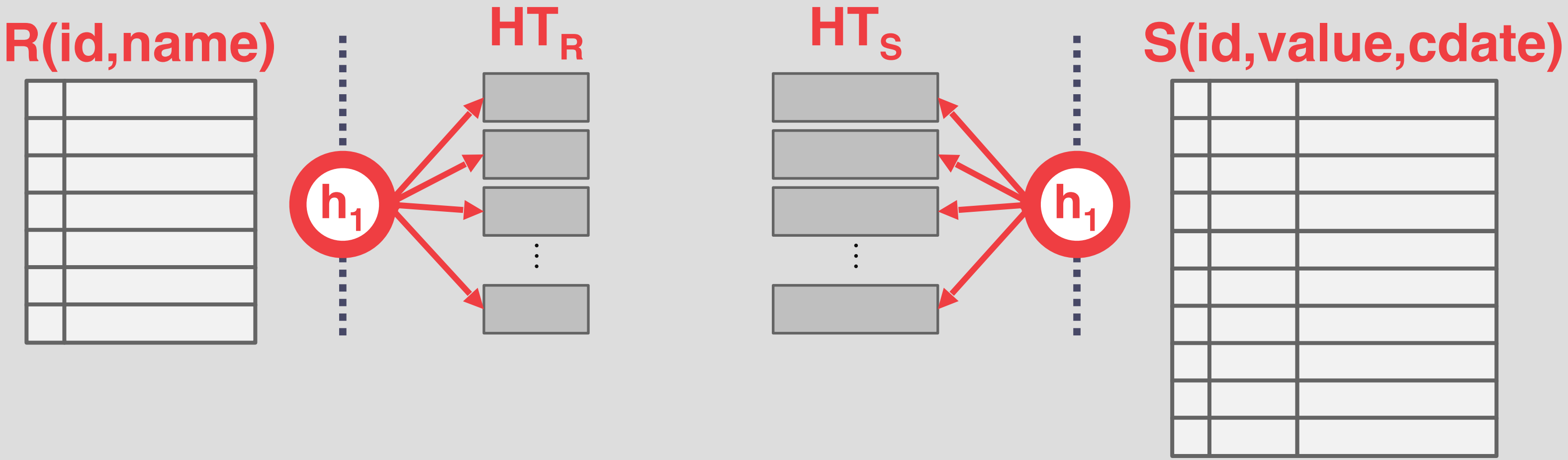
Hash **R** into (0, 1, ..., *max*) buckets.

Hash **S** into the same # of buckets with the same hash function.

PARTITIONED HASH JOIN

Hash **R** into (0, 1, ..., *max*) buckets.

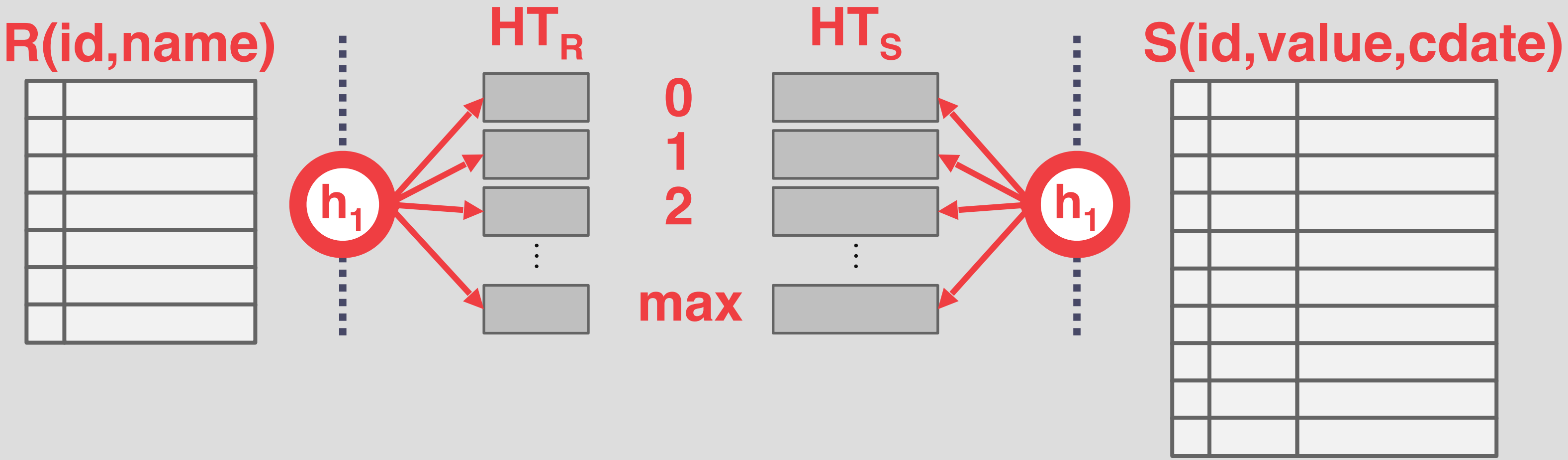
Hash **S** into the same # of buckets with the same hash function.



PARTITIONED HASH JOIN

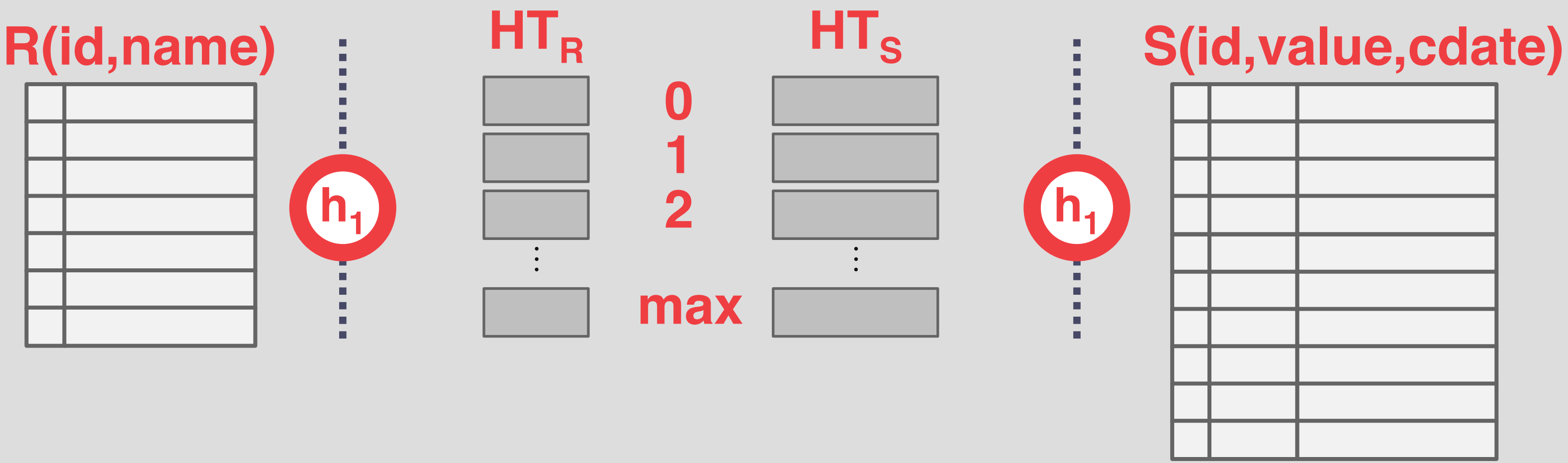
Hash **R** into (0, 1, ..., *max*) buckets.

Hash **S** into the same # of buckets with the same hash function.



PARTITIONED HASH JOIN

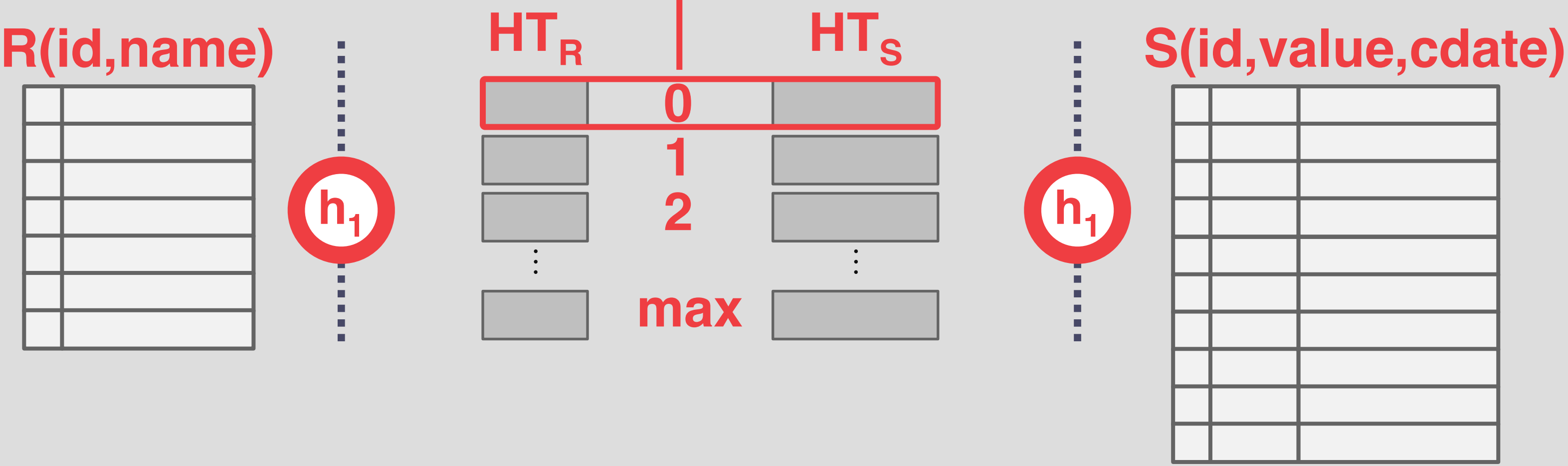
Perform regular hash join on each pair of matching buckets in the same level between **R** and **S**.



PARTITIONED HASH JOIN

Perform regular ha[...] matching buckets i[...] same level betwee[...] R and S.

build hash table $HT_{R,0}$ for $bucket_{R,0}$
foreach tuple $s \in bucket_{S,0}$
output, if $h_2(s) \in HT_{R,0}$



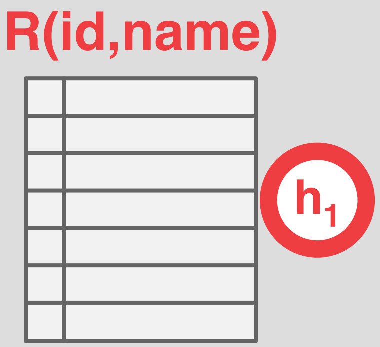
PARTITIONED HASH JOIN

If the buckets do not fit in memory, then use **recursive partitioning** to split the tables into chunks that will fit.

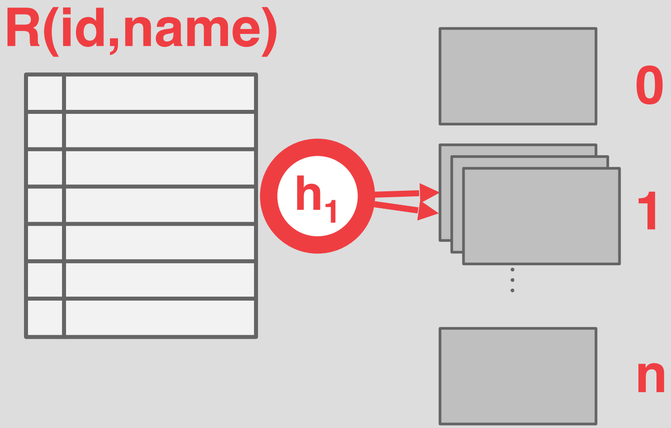
→ Build another hash table for **$bucket_{R,i}$** using hash function **h_2** (with **$h_2 \neq h_1$**).

→ Then probe it for each tuple of the other table's bucket at that level.

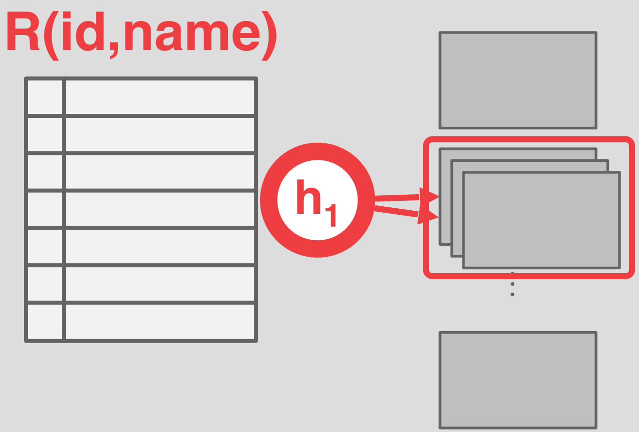
RECURSIVE PARTITIONING



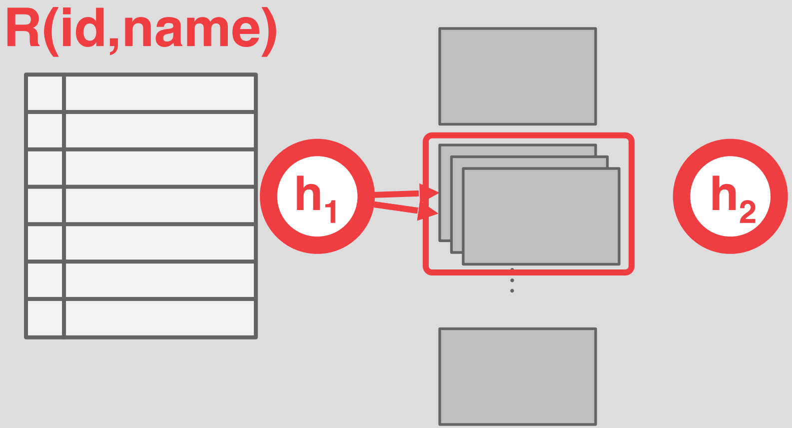
RECURSIVE PARTITIONING



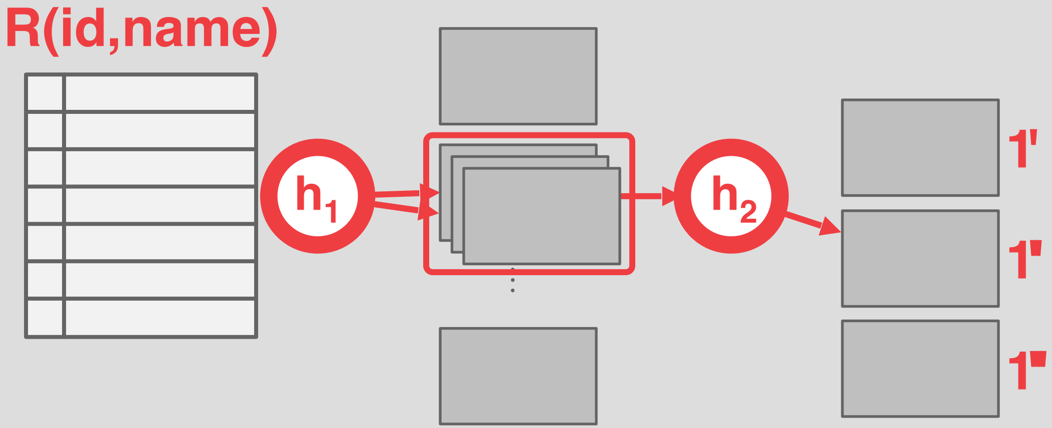
RECURSIVE PARTITIONING



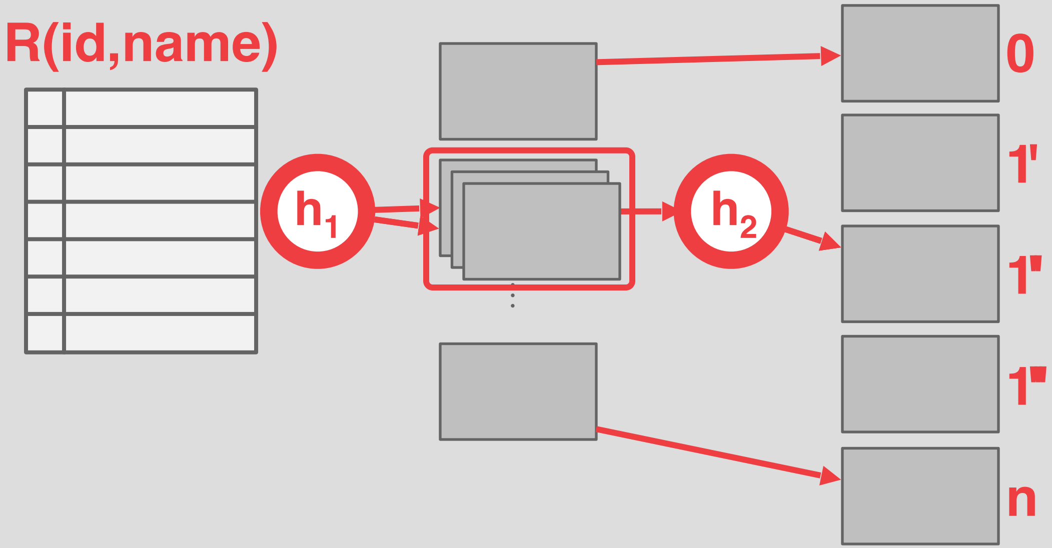
RECURSIVE PARTITIONING



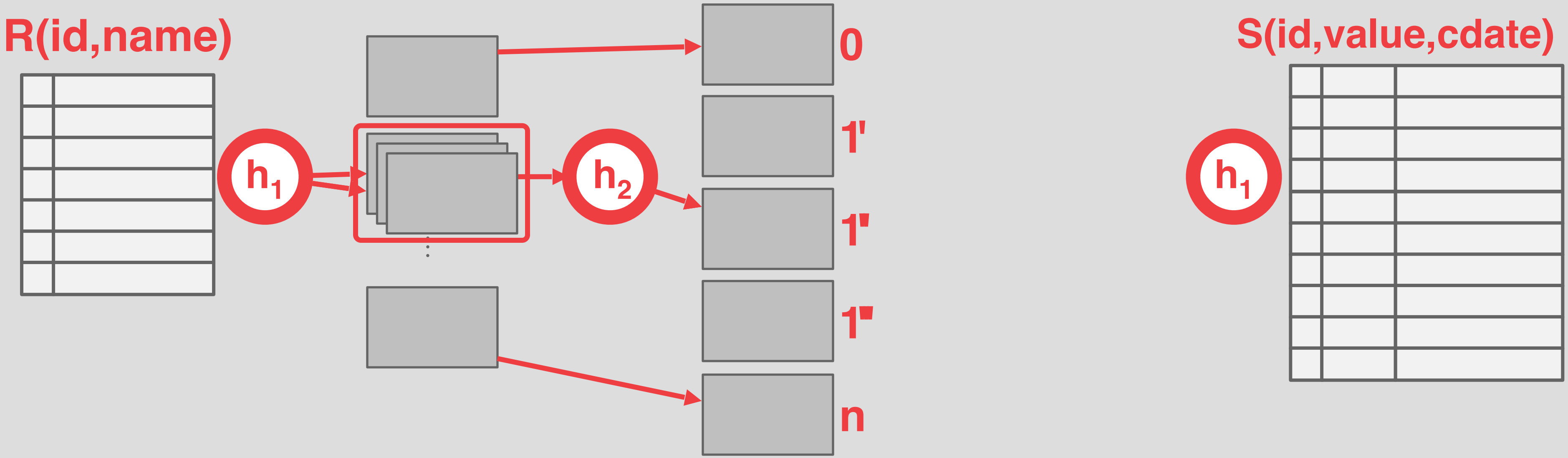
RECURSIVE PARTITIONING



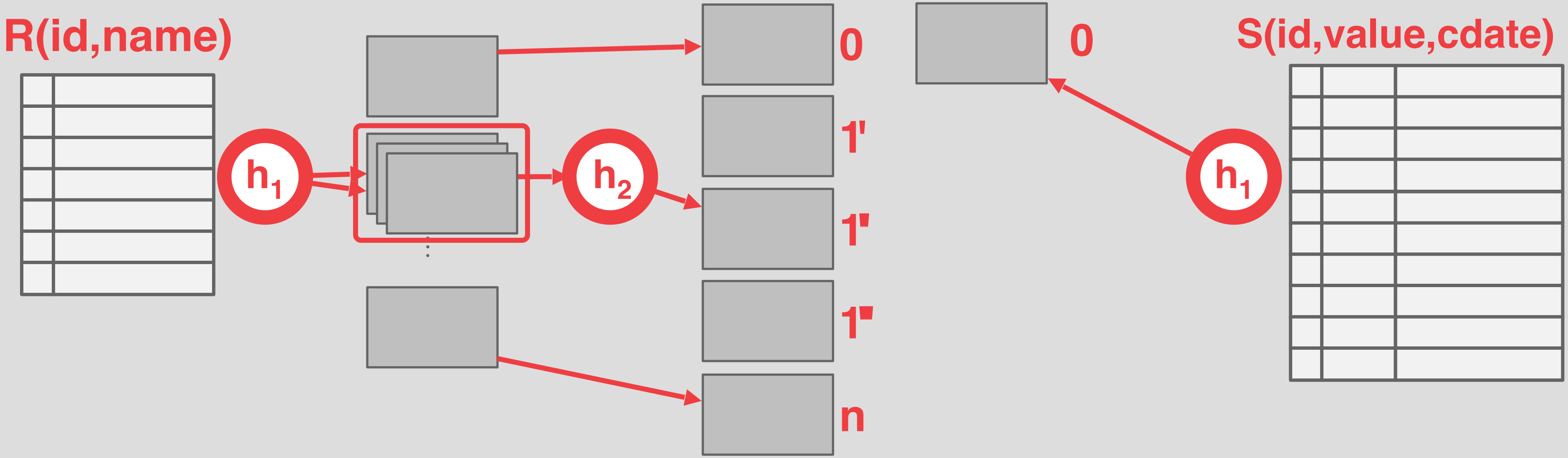
RECURSIVE PARTITIONING



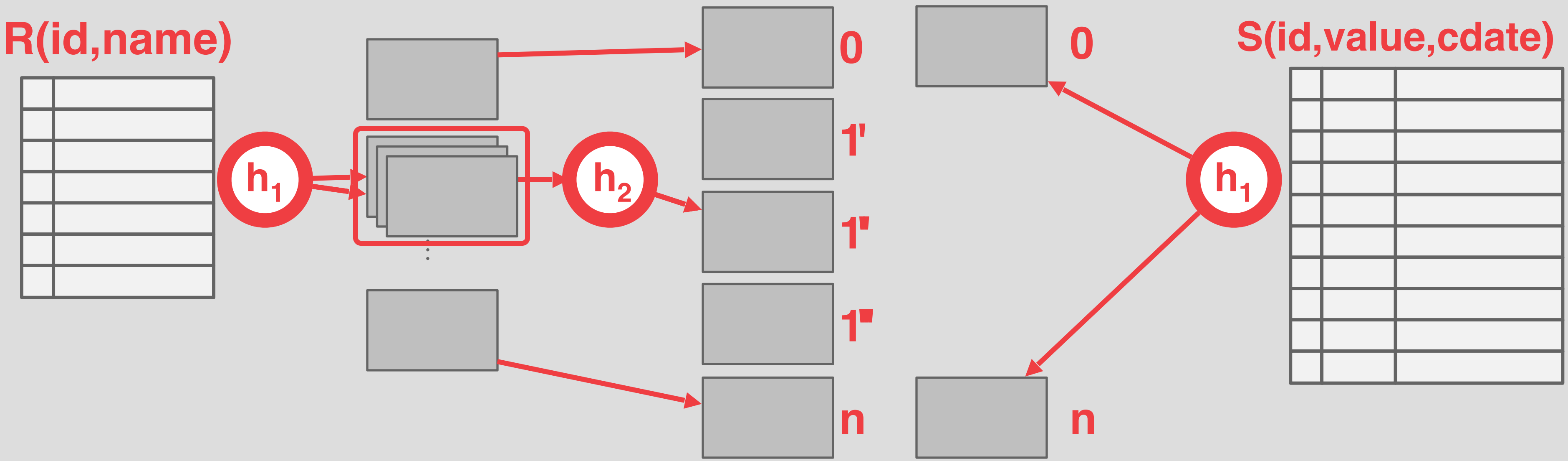
RECURSIVE PARTITIONING



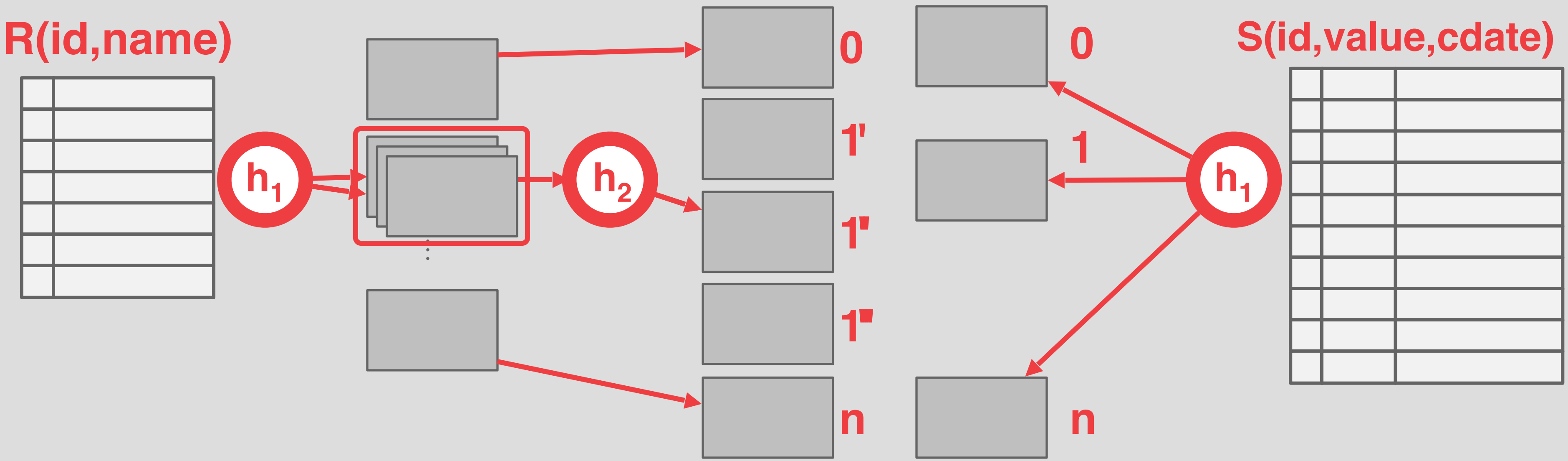
RECURSIVE PARTITIONING



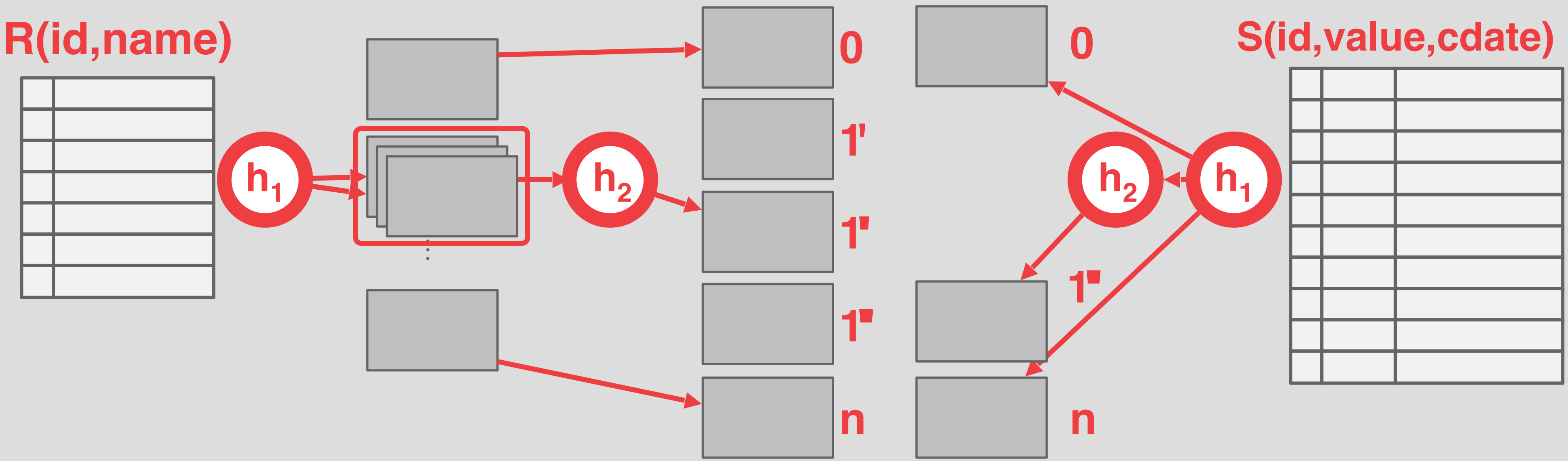
RECURSIVE PARTITIONING



RECURSIVE PARTITIONING



RECURSIVE PARTITIONING



PARTITIONED HASH JOIN

Cost of hash join?
- → Assume that we have enough buffers.
- → Cost: **$3(M + N)$**

PARTITIONED HASH JOIN

Cost of hash join?

→ Assume that we have enough buffers.

→ Cost: **3(M + N)**

Partitioning Phase:

→ Read+Write both tables

→ **2(M+N)** IOs

PARTITIONED HASH JOIN

Cost of hash join?
→ Assume that we have enough buffers.
→ Cost: **3(M + N)**

Partitioning Phase:
→ Read+Write both tables
→ **2(M+N)** IOs

Probing Phase:
→ Read both tables
→ **M+N** IOs

PARTITIONED HASH JOIN

Example database:

→ $M = 1000$, $m = 100{,}000$

→ $N = 500$, $n = 40{,}000$

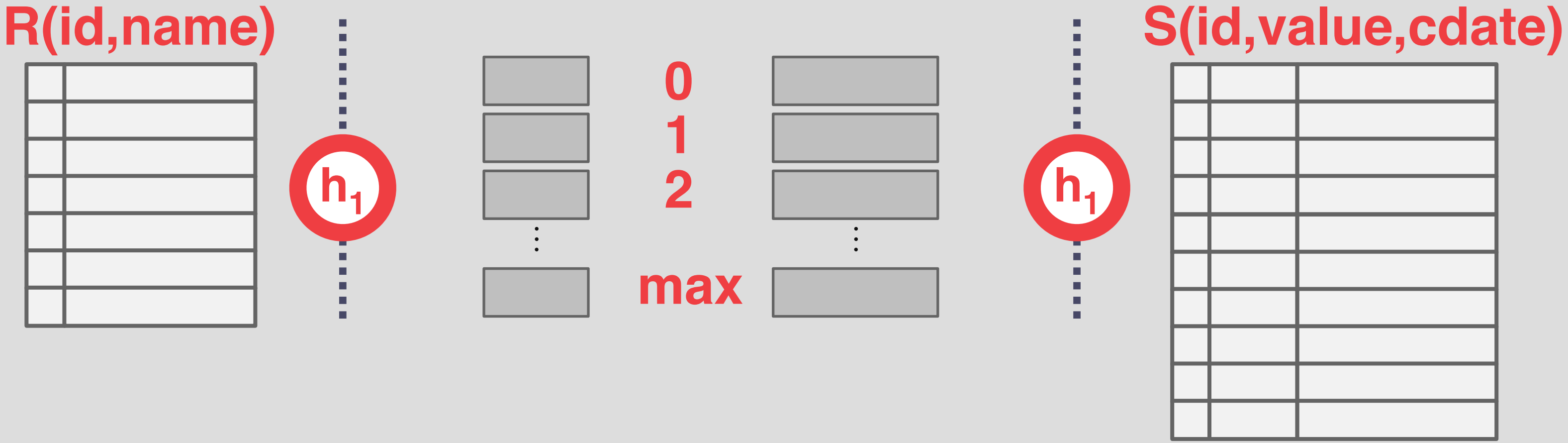
Cost Analysis:

→ $3 \cdot (M + N) = 3 \cdot (1000 + 500) = \mathbf{4{,}500\ IOs}$

→ At 0.1 ms/IO, Total time ≈ 0.45 seconds

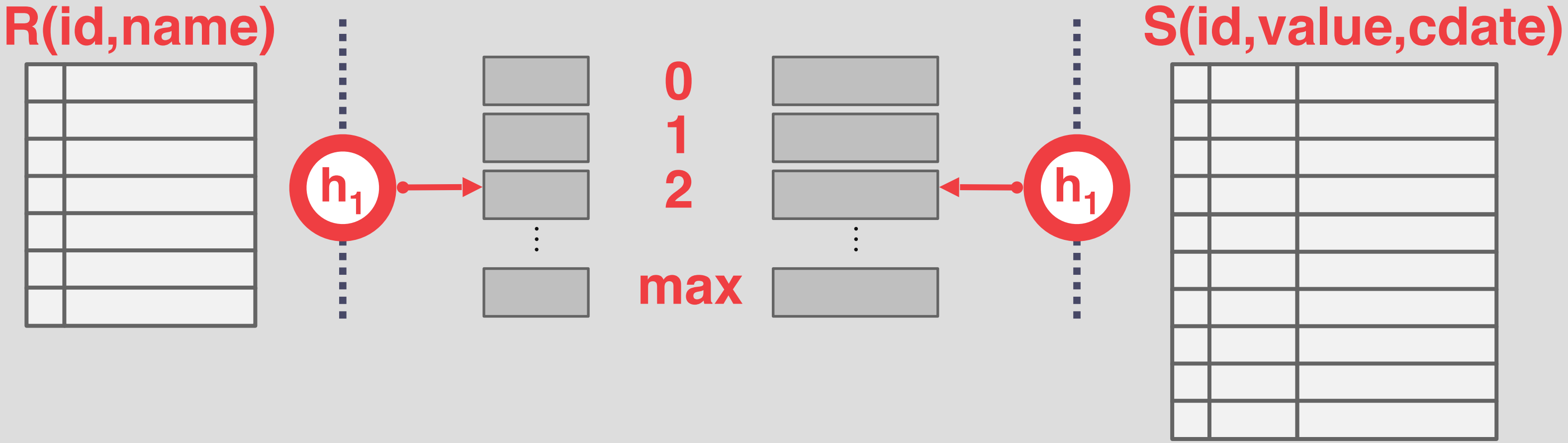
OPTIMIZATION: HYBRID HASH JOIN

If the keys are skewed, then the DBMS keeps the hot partition in-memory and immediately perform the comparison instead of spilling it to disk.
→ Difficult to get to work correctly. Rarely done in practice.



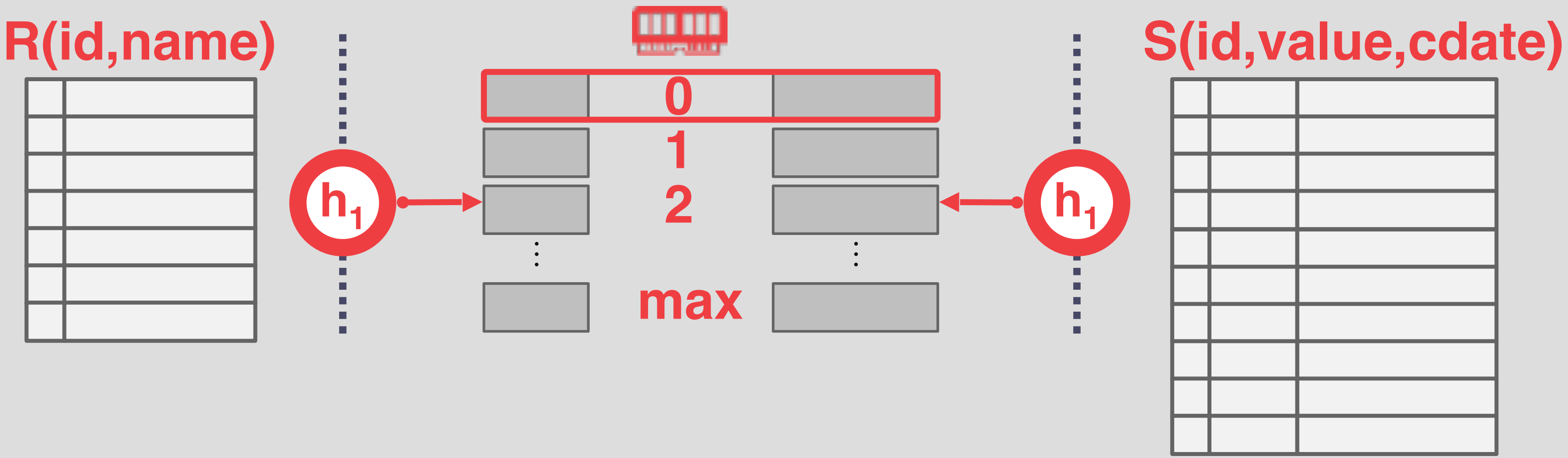
OPTIMIZATION: HYBRID HASH JOIN

If the keys are skewed, then the DBMS keeps the hot partition in-memory and immediately perform the comparison instead of spilling it to disk.
→ Difficult to get to work correctly. Rarely done in practice.



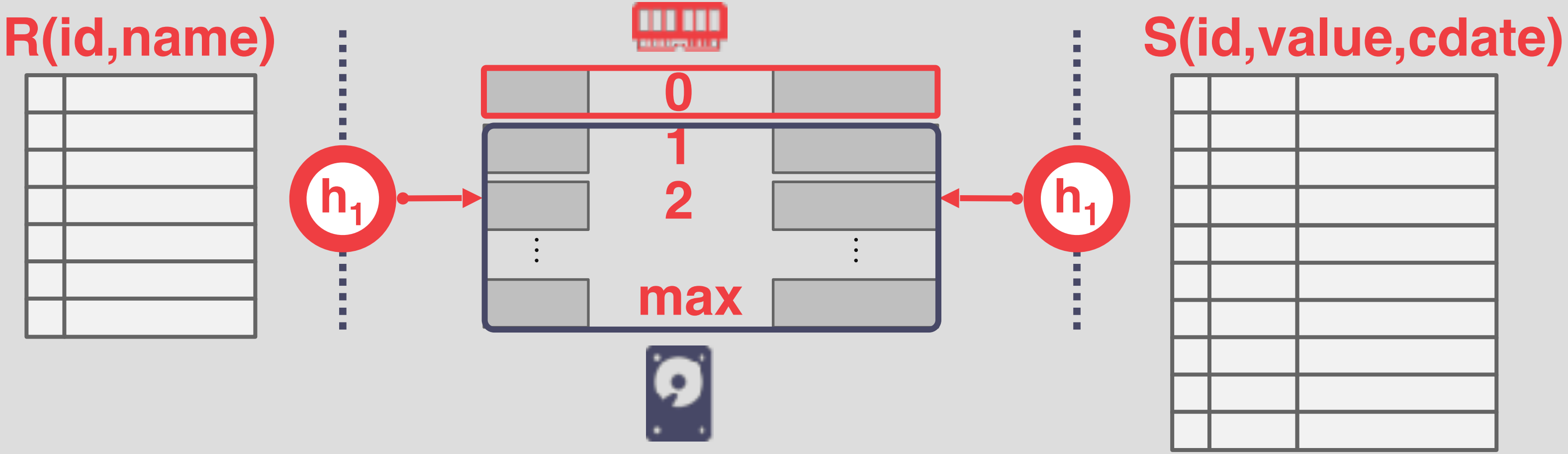
OPTIMIZATION: HYBRID HASH JOIN

If the keys are skewed, then the DBMS keeps the hot partition in-memory and immediately perform the comparison instead of spilling it to disk.
→ Difficult to get to work correctly. Rarely done in practice.



OPTIMIZATION: HYBRID HASH JOIN

If the keys are skewed, then the DBMS keeps the hot partition in-memory and immediately perform the comparison instead of spilling it to disk.
→ Difficult to get to work correctly. Rarely done in practice.



OBSERVATION

No constraint on the size of inner table.

If the DBMS knows the size of the outer table, then it can use a static hash table.
→ Less computational overhead for build / probe operations.

If we do not know the size, then we must use a dynamic hash table or allow for overflow pages.

JOIN ALGORITHMS: SUMMARY

Algorithm	IO Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Index Nested Loop Join	$M + (m \cdot C)$	Variable
Sort-Merge Join	$M + N +$ (sort cost)	0.75 seconds
Hash Join	$3 \cdot (M + N)$	0.45 seconds

CONCLUSION

Hashing is almost always better than sorting for operator execution.

Caveats:
→ Sorting is better on non-uniform data.
→ Sorting is better when result needs to be sorted.

Good DBMSs use either (or both).

NEXT CLASS

Composing operators together to execute queries.