

Carnegie
Mellon
University

Intro to Database
Systems (15-445/645)

Lecture #11

Join Algorithms

SPRING 2024 >> Prof. Jignesh Patel



ADMINISTRIVIA

Homework #2 is due Wed Feb 16th @ 11:59pm

Project #2 is due Wed March 13th @ 11:59pm

Mid-Term Exam is on Wednesday Feb 28th

- During regular class time from 12:30-1:50pm
- Please contact us if you need accommodations.
- Review session on Feb 26 during regular class time. Not recorded! Bring your questions to class – it will be an interactive review session.

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

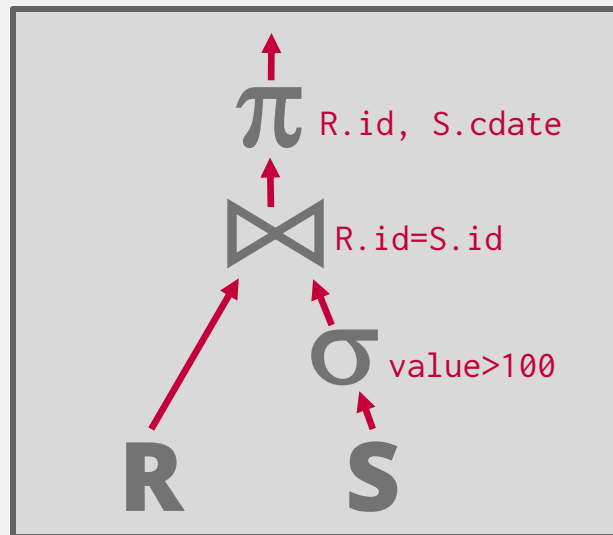
The operators are arranged in a tree.

Data flows from the leaves of the tree up towards the root.

→ We will discuss the granularity of the data movement next week.

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

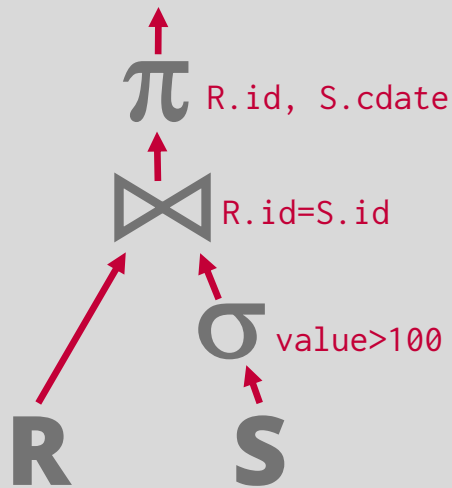
Decision #1: Output

→ 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
```



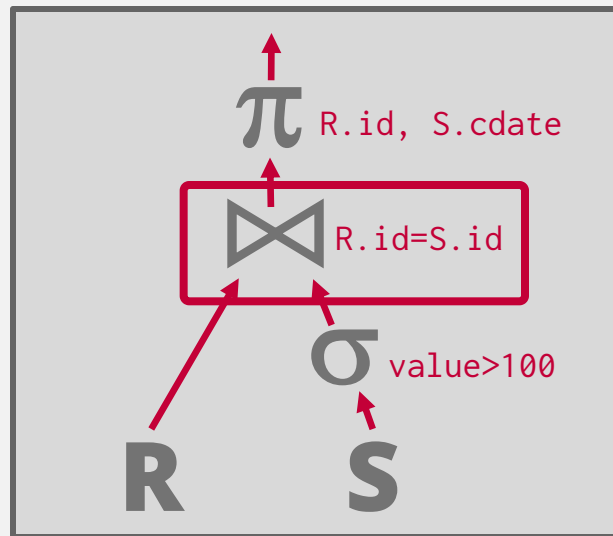
OPERATOR OUTPUT

For tuple $r \in R$ and tuple $s \in S$ 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.

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

R(id, name) **S(id, value, cdate)**

| id | name | | id | value | cdate |
|-----|------|---|-----|-------|---------|
| 123 | abc | ⋈ | 123 | 1000 | 2/20/24 |
| | | | 123 | 2000 | 2/20/24 |

| R.id | R.name | S.id | S.value | S.cdate |
|------|--------|------|---------|---------|
| 123 | abc | 123 | 1000 | 2/20/24 |
| 123 | abc | 123 | 2000 | 2/20/24 |

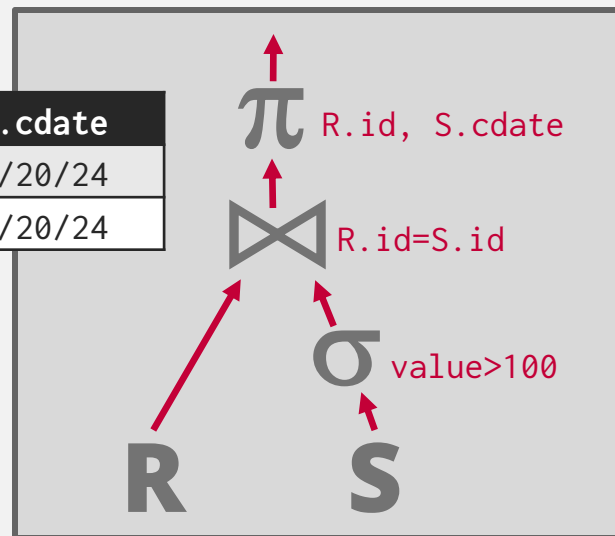
OPERATOR OUTPUT: DATA

Early Materialization:

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

| R.id | R.name | S.id | S.value | S.cdate |
|------|--------|------|---------|---------|
| 123 | abc | 123 | 1000 | 2/20/24 |
| 123 | abc | 123 | 2000 | 2/20/24 |

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



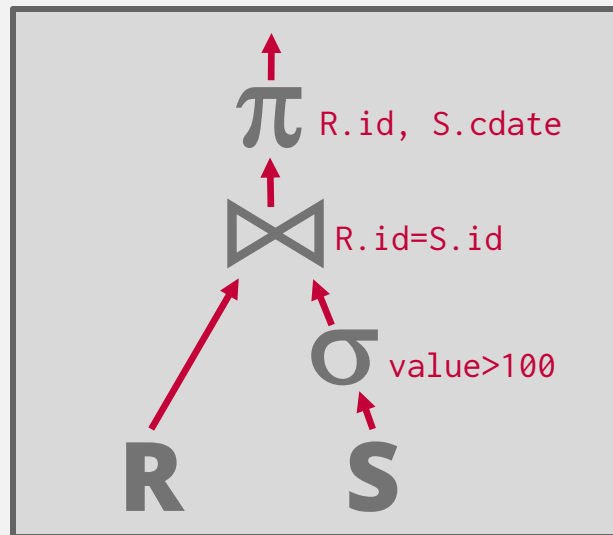
OPERATOR OUTPUT: DATA

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
```



OPERATOR OUTPUT: RECORD IDS

Late Materialization:

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

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

R(id, name) S(id, value, cdate)

| id | name | | id | value | cdate |
|-----|------|---|-----|-------|---------|
| 123 | abc | ⋈ | 123 | 1000 | 2/20/24 |
| | | | 123 | 2000 | 2/20/24 |

| R.id | R.RID | S.id | S.RID |
|------|-------|------|-------|
| 123 | R.### | 123 | S.### |
| 123 | R.### | 123 | S.### |

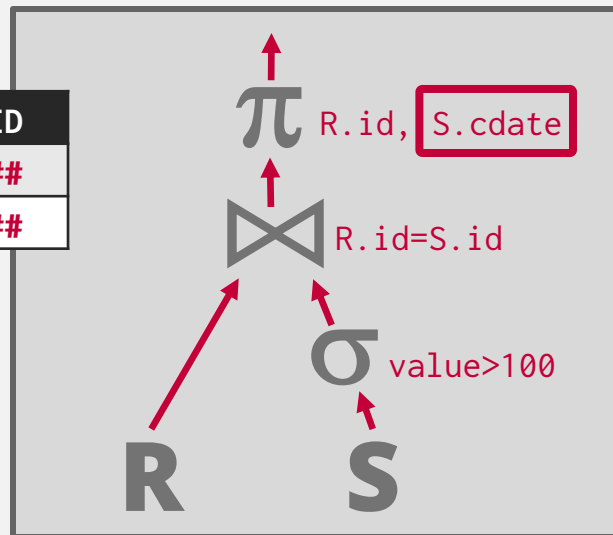
OPERATOR OUTPUT: RECORD IDS

Late Materialization:

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

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

| R.id | R.RID | S.id | S.RID |
|------|-------|------|-------|
| 123 | R.### | 123 | S.### |
| 123 | R.### | 123 | S.### |



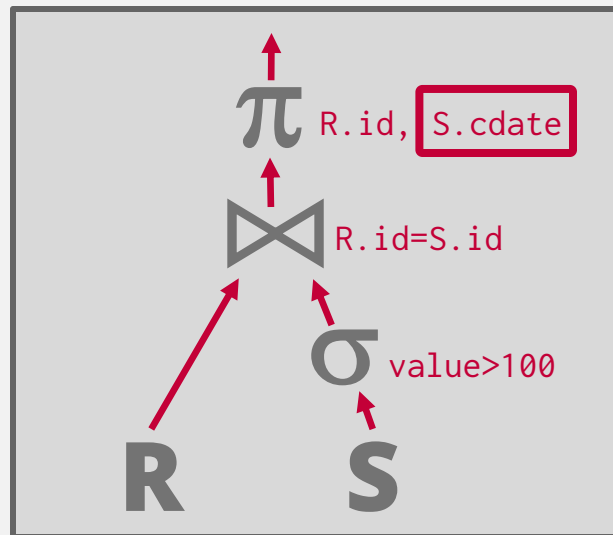
OPERATOR OUTPUT: RECORD IDS

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
```



COST ANALYSIS CRITERIA

Assume:

- M pages in table R , m tuples in R
- N pages in table S , n tuples in S

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

Cost Metric: # of I/Os to compute join

We ignore overall output costs because it depends on the data and is the same for all algorithms .

JOIN VS CROSS-PRODUCT

$R \bowtie S$ is the most common operation and thus must be carefully optimized.

$R \times S$ 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

- Naïve
- Block
- Index

Sort-Merge Join

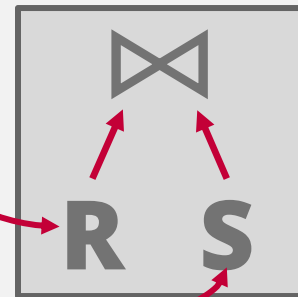
Hash Join

- Simple
- GRACE (Externally Partitioned)
- Hybrid

NAÏVE NESTED LOOP JOIN

```

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



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 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

NAÏVE NESTED LOOP JOIN

Why is this algorithm bad?

→ For every tuple 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 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

N pages
 n tuples

NAÏVE 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:

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

→ At 0.1 ms/IO, Total time \approx 1.3 hours

What if smaller table (**S**) is used as the outer table?

→ $N + (n \cdot M) = 500 + (40000 \cdot 1000) = 40,000,500$ IOs

→ At 0.1 ms/IO, Total time \approx 1.1 hours

BLOCK NESTED LOOP JOIN

```

foreach block  $B_R \in R$ :
  foreach block  $B_S \in S$ :
    foreach tuple  $r \in B_R$ :
      foreach tuple  $s \in B_S$ :
        if  $r$  and  $s$  match then emit
  
```

$R(id, name)$

| 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 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

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.

| <i>M</i> pages <i>m</i> tuples | R(id, name) | <table border="1" style="border-collapse: collapse; width: 100%; text-align: left;"> <thead> <tr style="background-color: #333; color: white;"> <th style="padding: 5px;">id</th> <th style="padding: 5px;">name</th> </tr> </thead> <tbody> <tr><td style="padding: 5px;">600</td><td style="padding: 5px;">MethodMan</td></tr> <tr><td style="padding: 5px;">200</td><td style="padding: 5px;">GZA</td></tr> <tr><td style="padding: 5px;">100</td><td style="padding: 5px;">Andy</td></tr> <tr><td style="padding: 5px;">300</td><td style="padding: 5px;">ODB</td></tr> <tr><td style="padding: 5px;">500</td><td style="padding: 5px;">RZA</td></tr> <tr><td style="padding: 5px;">700</td><td style="padding: 5px;">Ghostface</td></tr> <tr><td style="padding: 5px;">400</td><td style="padding: 5px;">Raekwon</td></tr> </tbody> </table> | id | name | 600 | MethodMan | 200 | GZA | 100 | Andy | 300 | ODB | 500 | RZA | 700 | Ghostface | 400 | Raekwon | | S(id, value, cdate) | <table border="1" style="border-collapse: collapse; width: 100%; text-align: left;"> <thead> <tr style="background-color: #333; color: white;"> <th style="padding: 5px;">id</th> <th style="padding: 5px;">value</th> <th style="padding: 5px;">cdate</th> </tr> </thead> <tbody> <tr><td style="padding: 5px;">100</td><td style="padding: 5px;">2222</td><td style="padding: 5px;">2/20/24</td></tr> <tr><td style="padding: 5px;">500</td><td style="padding: 5px;">7777</td><td style="padding: 5px;">2/20/24</td></tr> <tr><td style="padding: 5px;">400</td><td style="padding: 5px;">6666</td><td style="padding: 5px;">2/20/24</td></tr> <tr><td style="padding: 5px;">100</td><td style="padding: 5px;">9999</td><td style="padding: 5px;">2/20/24</td></tr> <tr><td style="padding: 5px;">200</td><td style="padding: 5px;">8888</td><td style="padding: 5px;">2/20/24</td></tr> </tbody> </table> | id | value | cdate | 100 | 2222 | 2/20/24 | 500 | 7777 | 2/20/24 | 400 | 6666 | 2/20/24 | 100 | 9999 | 2/20/24 | 200 | 8888 | 2/20/24 | <i>N</i> pages <i>n</i> tuples |
|-----------------------------------|-------------|---|----|------|-----|-----------|-----|-----|-----|------|-----|-----|-----|-----|-----|-----------|-----|---------|--|---------------------|---|----|-------|-------|-----|------|---------|-----|------|---------|-----|------|---------|-----|------|---------|-----|------|---------|-----------------------------------|
| id | name | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 600 | MethodMan | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 200 | GZA | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 100 | Andy | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 300 | ODB | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 500 | RZA | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 700 | Ghostface | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 400 | Raekwon | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| id | value | cdate | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 100 | 2222 | 2/20/24 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 500 | 7777 | 2/20/24 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 400 | 6666 | 2/20/24 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 100 | 9999 | 2/20/24 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 200 | 8888 | 2/20/24 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

BLOCK NESTED LOOP JOIN

Example database:

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

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

Cost Analysis:

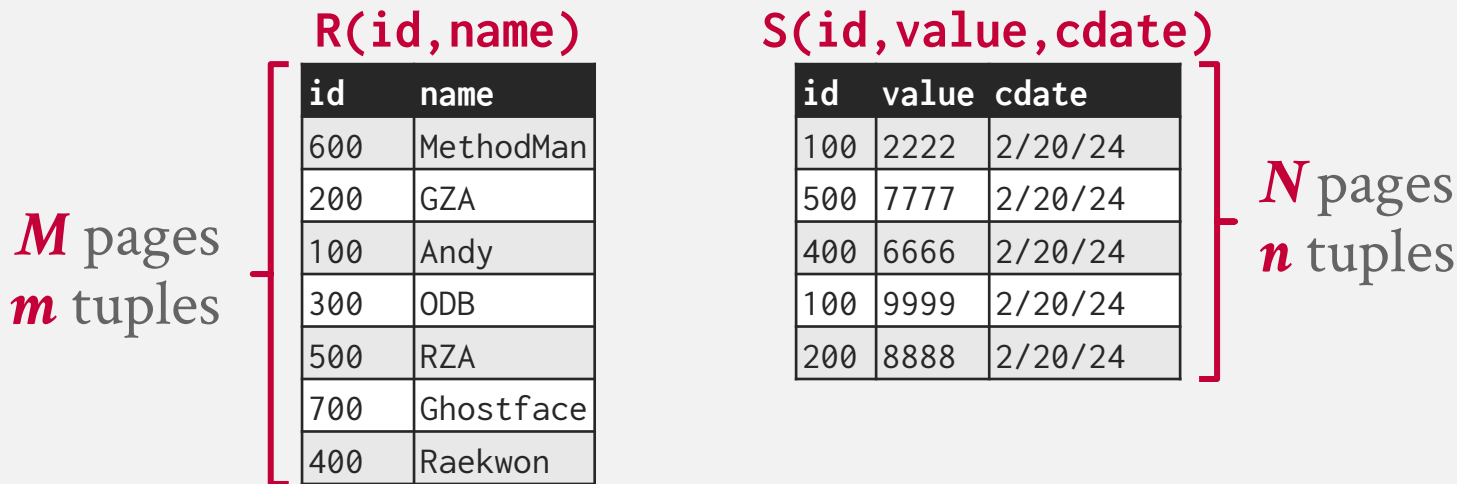
→ $M + (M \cdot N) = 1000 + (1000 \cdot 500) = 501,000$ IOs

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

BLOCK NESTED LOOP JOIN

If we have B buffers available:

- Use $B-2$ buffers for each block of the outer table.
- Use one buffer for the inner table, one buffer for output.



BLOCK NESTED LOOP JOIN

```

foreach  $B - 2$  pages  $p_R \in R$ :
  foreach page  $p_S \in S$ :
    foreach tuple  $r \in B - 2$  pages:
      foreach tuple  $s \in p_S$ :
        if  $r$  and  $s$  match then emit
  
```

$R(id, name)$

| 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 | cdates |
|-----|-------|---------|
| 100 | 2222 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

N pages
 n tuples

BLOCK NESTED LOOP JOIN

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

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

If the outer relation fits in memory ($M < B-2$):

→ **Cost:** $M + N = 1000 + 500 = 1500$ I/Os

→ At 0.1ms per I/O, Total time ≈ 0.15 seconds

If we have $B=102$ buffer pages:

→ **Cost:** $M + (\lceil M / (B-2) \rceil \cdot N) = 1000 + 10 \cdot 500 = 6000$ I/Os

→ Or can switch inner/outer relations, giving us cost: $500 + 5 \cdot 1000 = 5500$ I/Os
 ≈ 0.55 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.

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  $r \in R$ :
  foreach tuple  $s \in \text{Index}(r_i = s_j)$ :
    if  $r$  and  $s$  match then emit
  
```

M pages
 m tuples

$R(\text{id}, \text{name})$

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

$S(\text{id}, \text{value}, \text{cdate})$

| id | value | cdate |
|-----|-------|---------|
| 100 | 2222 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

$\text{Index}(S.\text{id})$



N pages
 n tuples

INDEX NESTED LOOP JOIN

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

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

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 | cdates |
|-----|-------|---------|
| 100 | 2222 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

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

- Naïve
- Block
- Index

SORT-MERGE JOIN

Phase #1: Sort

- Sort both tables on the join key(s).
- You can use any appropriate sort algorithm
- These phases are distinct from the sort/merge phases of an external merge sort, from the previous 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
cursorR ← Rsorted, cursorS ← Ssorted
while cursorR and cursorS:
    if cursorR > cursorS:
        increment cursorS
    if cursorR < cursorS:
        increment cursorR
        backtrack cursorS (if necessary)
    elif cursorR and cursorS match:
        emit
        increment cursorS
```


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 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |

```
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 | 2/20/24 |
| 500 | 7777 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |


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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |


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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |

```
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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |

```
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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |


```
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 | 2/20/24 |


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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |


```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |


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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |

```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 400 | Raekwon | 200 | 6666 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 400 | Raekwon | 200 | 6666 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 400 | Raekwon | 200 | 6666 | 2/20/24 |
| 500 | RZA | 500 | 7777 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |



```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 400 | Raekwon | 200 | 6666 | 2/20/24 |
| 500 | RZA | 500 | 7777 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |

```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 400 | Raekwon | 200 | 6666 | 2/20/24 |
| 500 | RZA | 500 | 7777 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |

```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 400 | Raekwon | 200 | 6666 | 2/20/24 |
| 500 | RZA | 500 | 7777 | 2/20/24 |

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 | 2/20/24 |
| 100 | 9999 | 2/20/24 |
| 200 | 8888 | 2/20/24 |
| 400 | 6666 | 2/20/24 |
| 500 | 7777 | 2/20/24 |

```
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 | 2/20/24 |
| 100 | Andy | 100 | 9999 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 200 | GZA | 200 | 8888 | 2/20/24 |
| 400 | Raekwon | 200 | 6666 | 2/20/24 |
| 500 | RZA | 500 | 7777 | 2/20/24 |

SORT-MERGE JOIN

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

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

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**) = $2000 \cdot (1 + \lceil \log_{99} 1000 / 100 \rceil) = 4000$ I/Os

→ Sort Cost (**S**) = $1000 \cdot (1 + \lceil \log_{99} 500 / 100 \rceil) = 2000$ I/Os

→ Merge Cost = $(1000 + 500) = 1500$ I/Os

→ Total Cost = $4000 + 2000 + 1500 = 7500$ I/Os

→ At 0.1 ms/IO, Total time ≈ 0.75 seconds

SORT-MERGE JOIN: THE ADVANCED VERSION

Merge the run-generation phase and the join phase.

- Produce initial runs for R. These are $\approx B$ pages long.
- Produce initial runs for S. These are also $\approx B$ pages long.
- If there are enough pages to hold one page from each run in memory, do the join and merge passes together. Need $\lceil R / B \rceil + \lceil S / B \rceil$ pages
- Thus, in 2 passes (1 initial run creation, then sort-merge-join pass) you can compute the join.
- Some more modifications are possible, including making the initial runs be of size $2B$, which guarantees that a 2-pass sort-merge join can be done if $B > \sqrt{\text{MAX}(R, S)}$

SORT-MERGE JOIN: THE ADVANCED VERSION

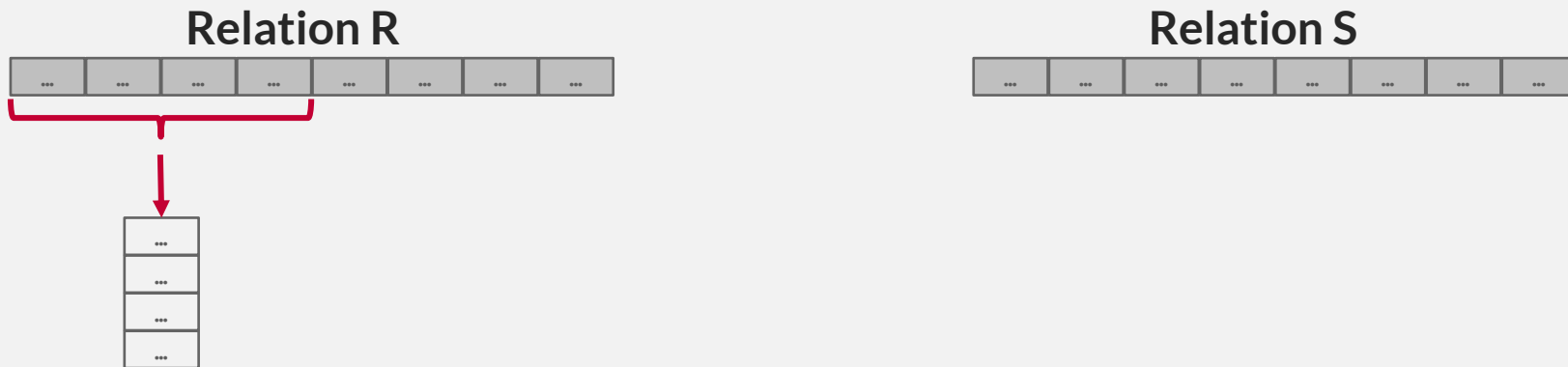
Relation R

| | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|
| ... | ... | ... | ... | ... | ... | ... | ... |
|-----|-----|-----|-----|-----|-----|-----|-----|

Relation S

| | | | | | | | |
|-----|-----|-----|-----|-----|-----|-----|-----|
| ... | ... | ... | ... | ... | ... | ... | ... |
|-----|-----|-----|-----|-----|-----|-----|-----|

SORT-MERGE JOIN: THE ADVANCED VERSION



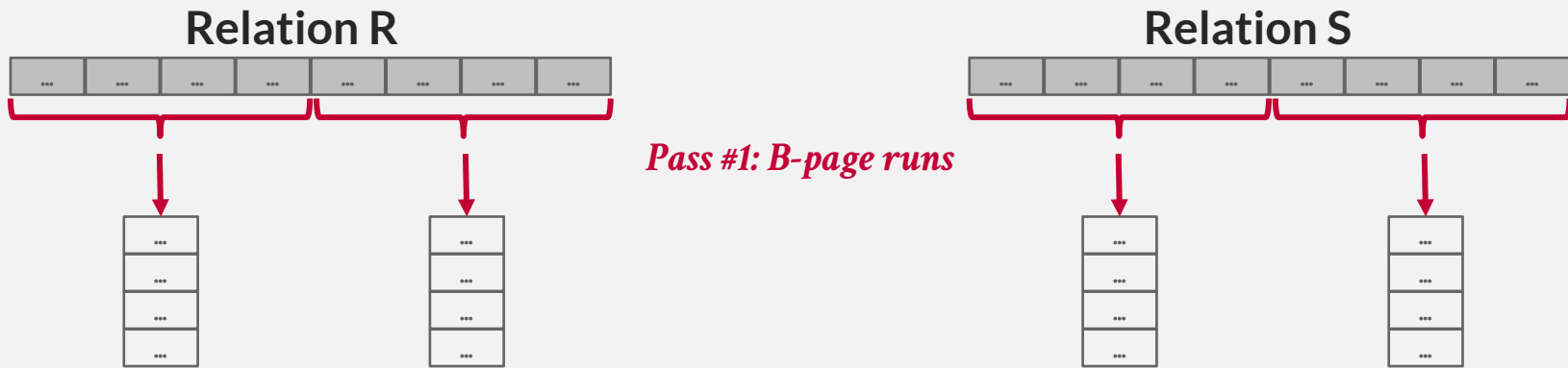
SORT-MERGE JOIN: THE ADVANCED VERSION



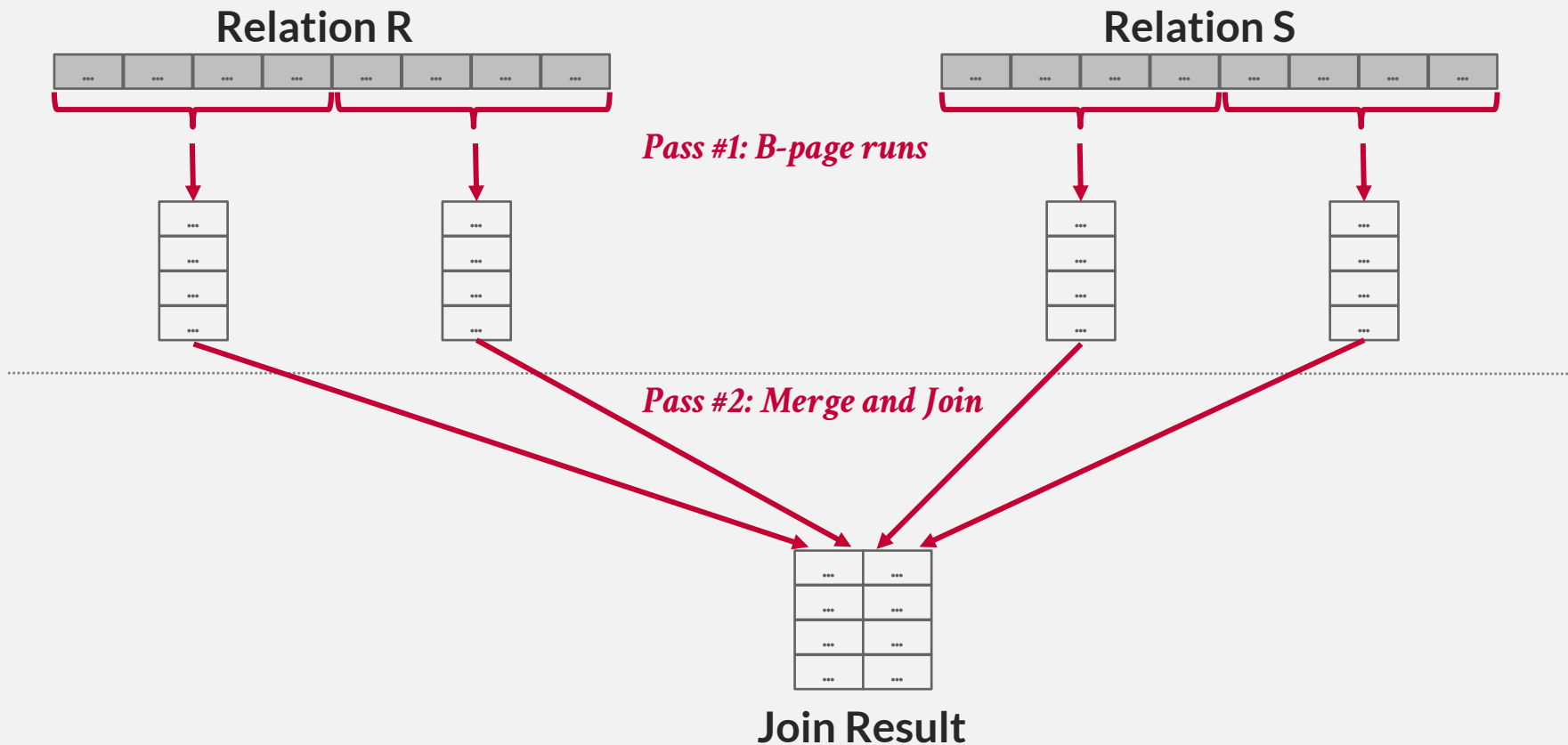
SORT-MERGE JOIN: THE ADVANCED VERSION



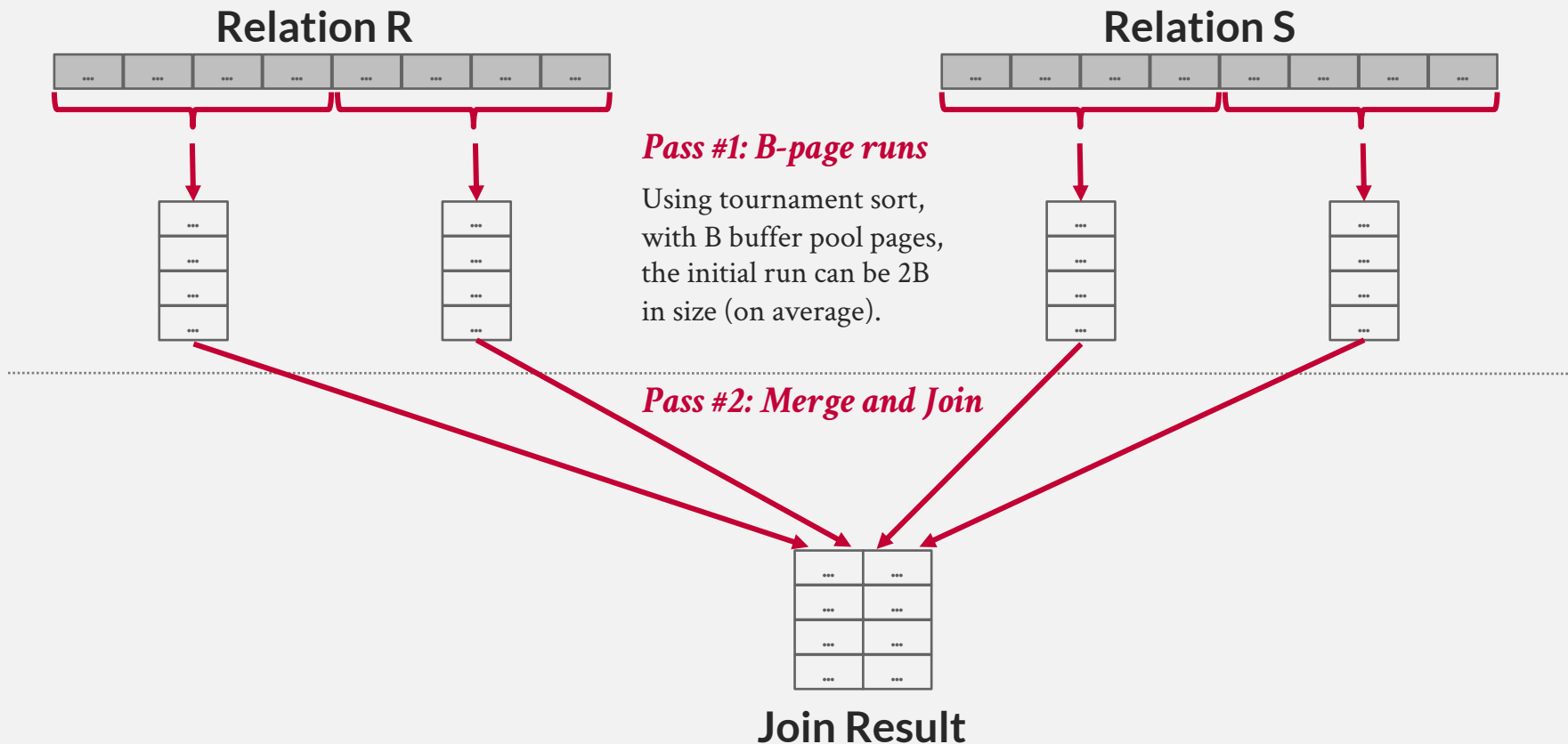
SORT-MERGE JOIN: THE ADVANCED VERSION



SORT-MERGE JOIN: THE ADVANCED VERSION



SORT-MERGE JOIN: THE ADVANCED VERSION



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: $(M \cdot N) + (\text{sort cost})$

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

If that value is hashed to some partition \mathbf{i} , the \mathbf{R} tuple must be in \mathbf{r}_i and the \mathbf{S} tuple in \mathbf{s}_i .

Therefore, \mathbf{R} tuples in \mathbf{r}_i need only to be compared with \mathbf{S} tuples in \mathbf{s}_i .

SIMPLE 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.

SIMPLE HASH JOIN ALGORITHM

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

$R(id, name)$

| | |
|--|--|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

$S(id, value, cdate)$

| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

SIMPLE HASH JOIN ALGORITHM

```

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

$R(id, name)$

| | |
|--|--|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

Hash Table

HT_R

| |
|---|
| |
| |
| |
| ⋮ |
| |

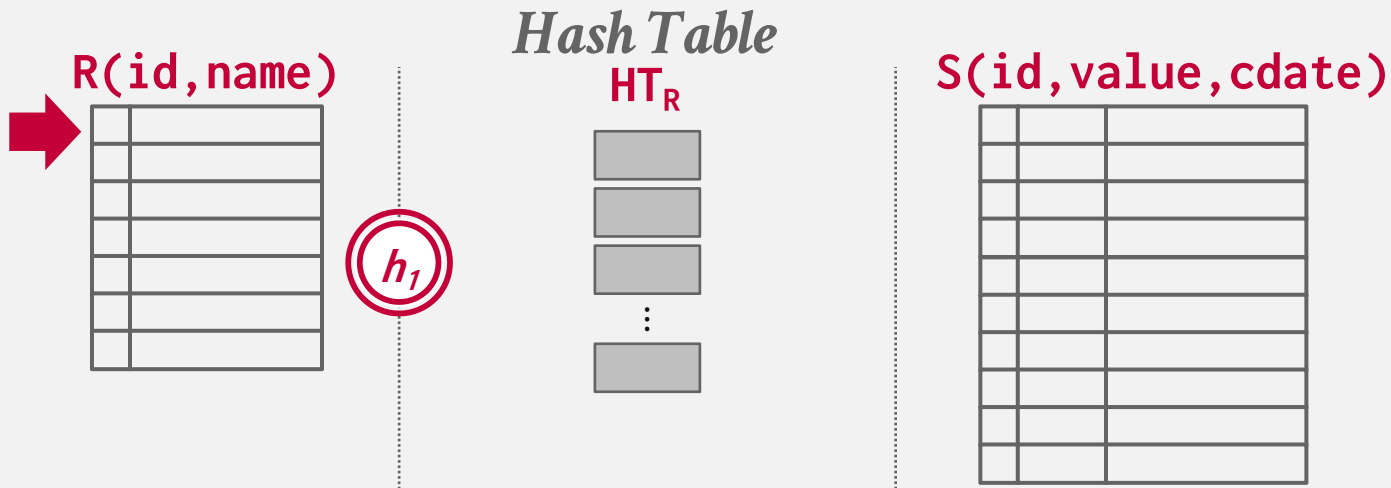
$S(id, value, cdate)$

| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

SIMPLE HASH JOIN ALGORITHM

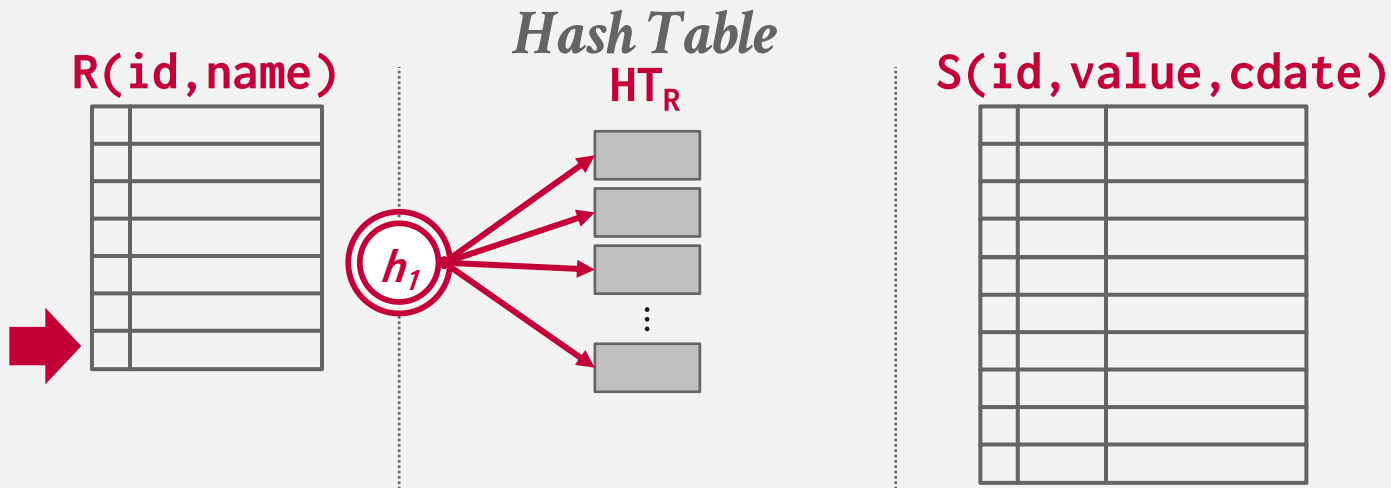
```

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



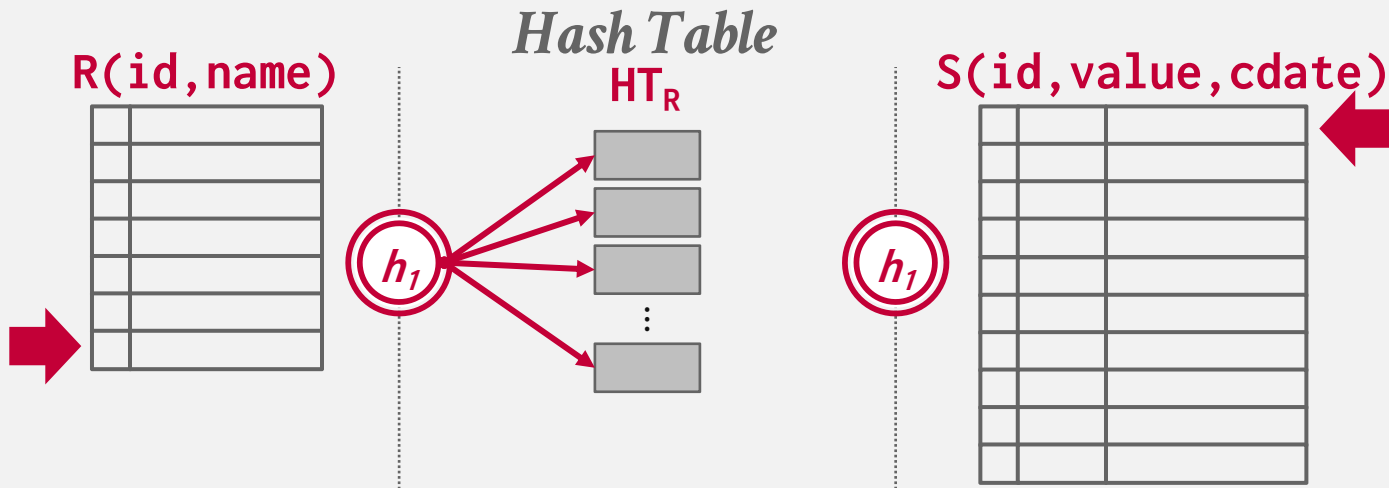
SIMPLE HASH JOIN ALGORITHM

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



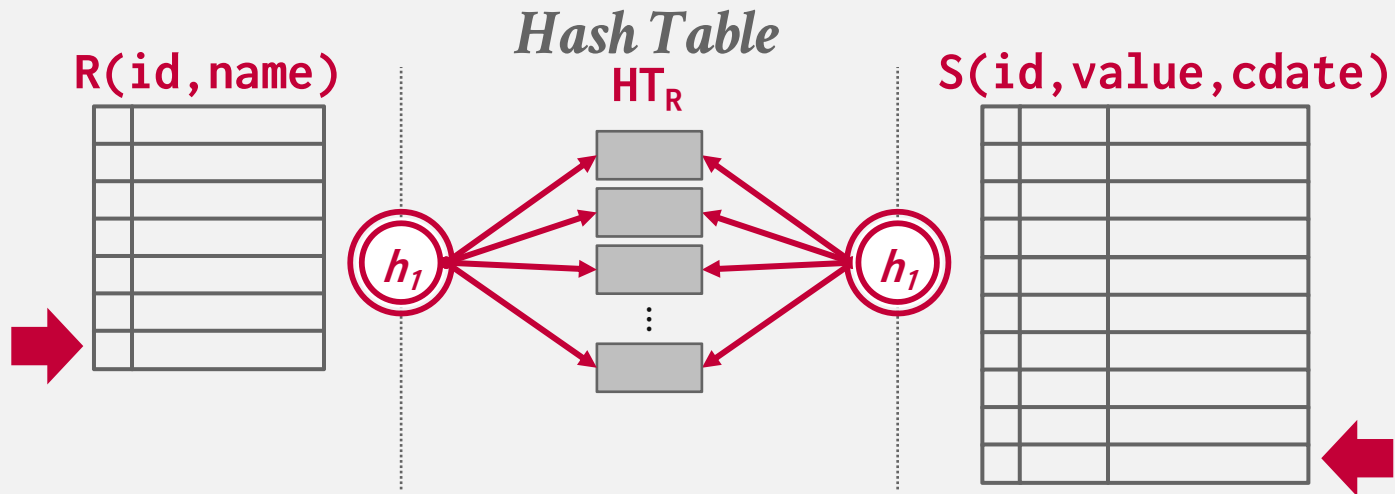
SIMPLE HASH JOIN ALGORITHM

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



SIMPLE 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

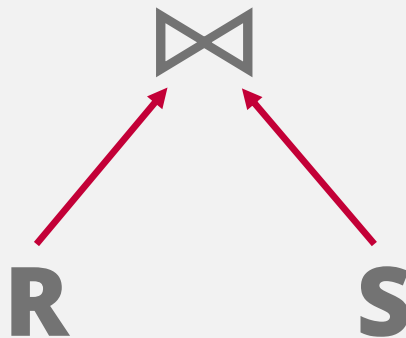
Key: The attribute(s) that the query is joining on
→ The hash table needs to store the key to verify that we have a correct match, in case of hash collisions.

Value: It varies per DBMS
→ Depends on what the next query operators will do with the output from the join
→ Early vs. Late Materialization

OPTIMIZATION: PROBE FILTER

Create a probe filter (such as a Bloom Filter) during the build phase if the key is likely to not exist in the inner relation

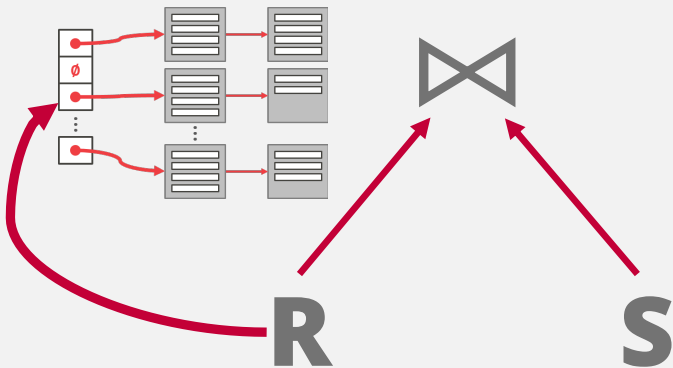
- Check the filter before probing the hash table
- Fast because the filter fits in CPU cache
- Sometimes called *sideways information passing*.



OPTIMIZATION: PROBE FILTER

Create a probe filter (such as a Bloom Filter) during the build phase if the key is likely to not exist in the inner relation

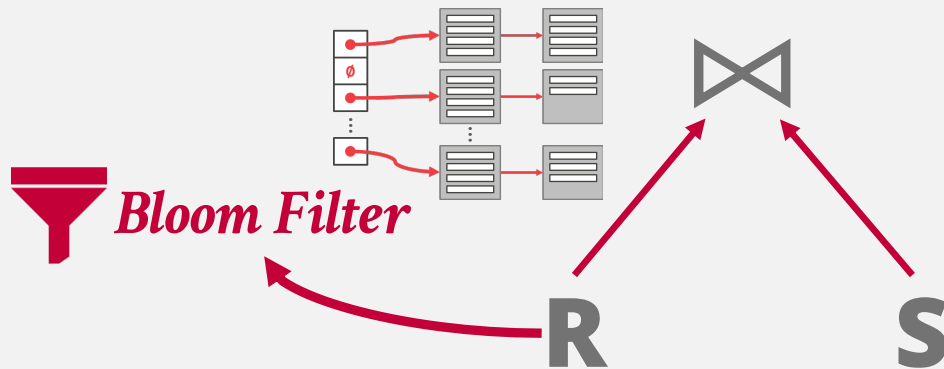
- Check the filter before probing the hash table
- Fast because the filter fits in CPU cache
- Sometimes called *sideways information passing*.



OPTIMIZATION: PROBE FILTER

Create a probe filter (such as a Bloom Filter) during the build phase if the key is likely to not exist in the inner relation

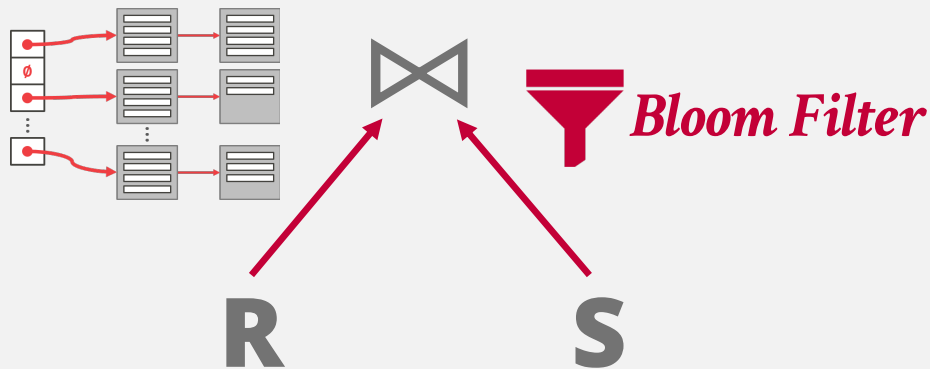
- Check the filter before probing the hash table
- Fast because the filter fits in CPU cache
- Sometimes called *sideways information passing*.



OPTIMIZATION: PROBE FILTER

Create a probe filter (such as a Bloom Filter) during the build phase if the key is likely to not exist in the inner relation

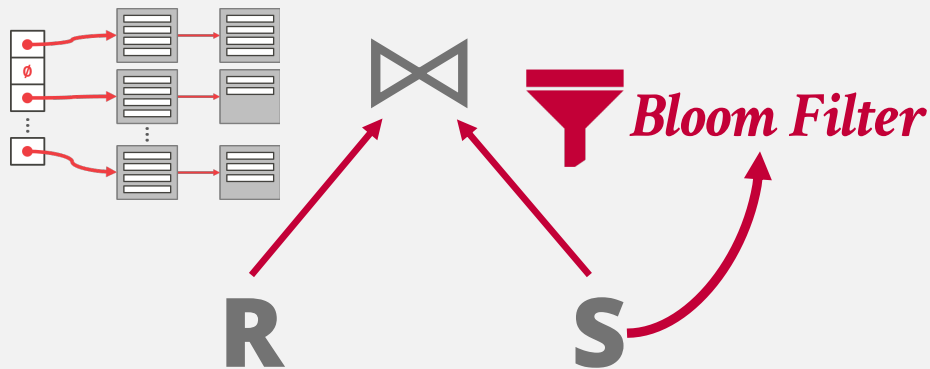
- Check the filter before probing the hash table
- Fast because the filter fits in CPU cache
- Sometimes called *sideways information passing*.



OPTIMIZATION: PROBE FILTER

Create a probe filter (such as a Bloom Filter) during the build phase if the key is likely to not exist in the inner relation

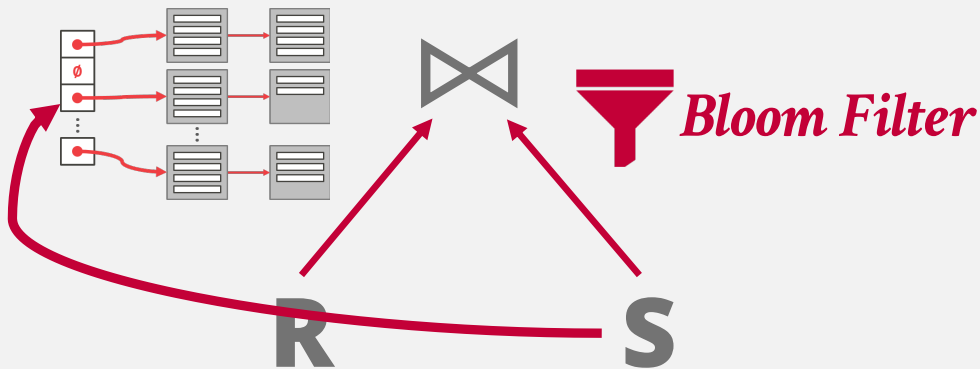
- Check the filter before probing the hash table
- Fast because the filter fits in CPU cache
- Sometimes called *sideways information passing*.



OPTIMIZATION: PROBE FILTER

Create a probe filter (such as a Bloom Filter) during the build phase if the key is likely to not exist in the inner relation

- Check the filter before probing the hash table
- Fast because the filter fits in CPU cache
- Sometimes called *sideways information passing*.



HASH JOINS OF LARGE RELATIONS

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.

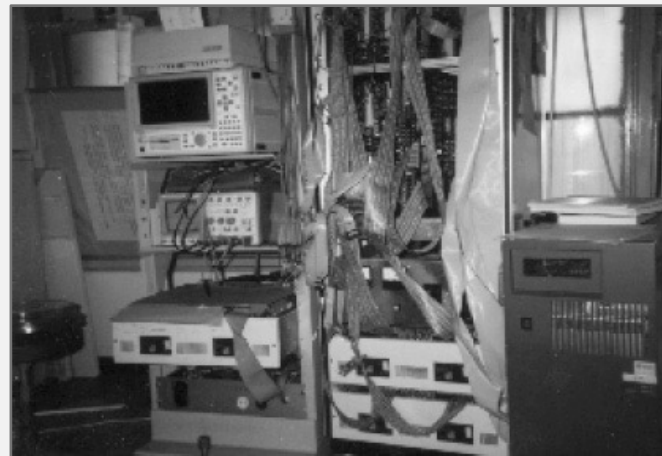
PARTITIONED HASH JOIN

Hash join when tables do not fit in memory.

- **Partition 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

Britton-Lee's technical achievements have created the Intelligent Data Base Machine, oriented to managers who know the value of a responsive information system. Truly user-oriented—even to

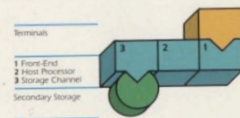
people without programming knowledge—the IDM 500 provides some remarkable advantages. Imagine how the features described inside can improve YOUR company's information productivity...

NOW

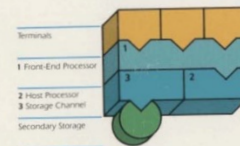


The IDM 500 A Logical Development

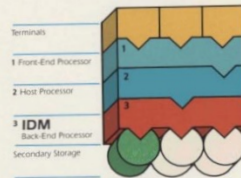
As data systems have evolved, the presence of special-purpose elements has become increasingly important, as these diagrams will illustrate:



In the 1960's, a single central processing unit (CPU) was required to monitor time-sharing among terminal users; to batch process computing tasks, and to control the access to stored data.



Through the development of front-end communication processors, the workload on the CPU was reduced. It was then able to perform its basic task of data processing much more efficiently. But the task of managing the data base was still imposed upon it.



Now Britton-Lee's IDM 500 special-purpose, back-end data-base processor brings full efficiency to the host computer and intelligent terminals, so that they can properly perform their correct functions.



IBM DB2 Analytics Accelerator - GSE Management Summit

Choosing the best fit Key indicators



IBM Netezza

- Performance and Price/performance leader
- Speed and ease of deployment and administration

IBM Netezza standalone appliance

- Strategic requirement for standalone decision support system
- If primary data feeds are from distributed applications
- Deep analytics applications or in-database mining

IBM DB2 Analytics Accelerator for z/OS

- Transparent acceleration of existing reporting workload on DB2

Teradata IntelliFlex™ 100% Solid State Performance

Up to: **7.5x** Performance for Com Intensive Analytics

4.5x Performance for Data Warehouse Analytics

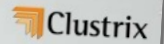
3.5x Data Capacity

2.0x Performance per kW



Note: comparisons to the previous generation IntelliFlex platform are on a per cabinet basis. Workloads will see up to this amount of benefit

CLUSTRIX APPLIANCE



Clustrix Appliance 3 Node Cluster (CLX 4110)

- 24 Intel Xeon CPU cores
- 144GB RAM
- 6GB NVRAM
- 1.35TB Intel SSD protected (2.7TB raw) data capacity

Complete Family Of Database Machines For OLTP, Data Warehousing & Consolidated Workloads

Oracle Exadata X2-2



Oracle Exadata X2-8



- Quarter, Half, Full and Multi-Racks

- Full and Multi-Racks

Choosing the best fit Key indicators



Yellowbrick Data Warehouse Architecture

Real-time Feeds
Ingest IoT or OLTP data
Capture 100,000s
of rows per second



Periodic Bulk Loads
Capture terabytes
of data, petabytes
over time



Load and Transform
Use existing ETL tools including
intensive push-down ELT



Interactive Applications
Serve short queries in
under 100 milliseconds



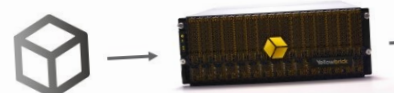
Powerful Analytics
Respond to
complex BI queries
in just a few seconds



Business Critical Reporting
Workload management
for prioritized responses



Source: yellowbrickdata.com



Teradata IntelliFlex 100% Solid State Performance

Up to:



4.5x Performance for Data Warehouse Analytics

3.5x Data Capacity

2.0x Performance per kW

Note: comparisons to the previous generation IntelliFlex platform are on a per cabinet basis. Workloads will see up to this amount of benefit.

de Cluster (CLX 4110)

Database Machines & Consolidated Workloads

Oracle Exadata X2-8



• Quarter, Half, Full and Multi-Racks

• Full and Multi-Racks

PARTITIONED HASH JOIN

Hash join when tables do not fit in memory.

- **Partition 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 PARTITION PHASE

Hash **R** into k buckets.

Hash **S** into k buckets with same hash function.

Write buckets to disk when they get full.

R(id, name)

| | |
|--|--|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

S(id, value, cdate)

| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |

PARTITIONED HASH JOIN PARTITION PHASE

Hash **R** into k buckets.

Hash **S** into k buckets with same hash function.

Write buckets to disk when they get full.

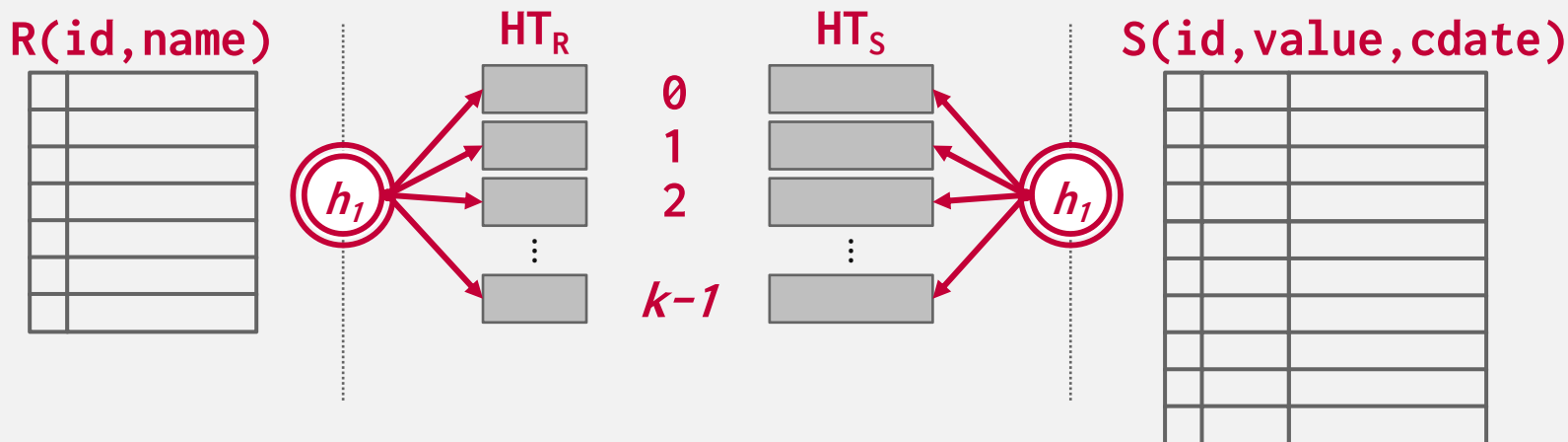


PARTITIONED HASH JOIN PARTITION PHASE

Hash **R** into k buckets.

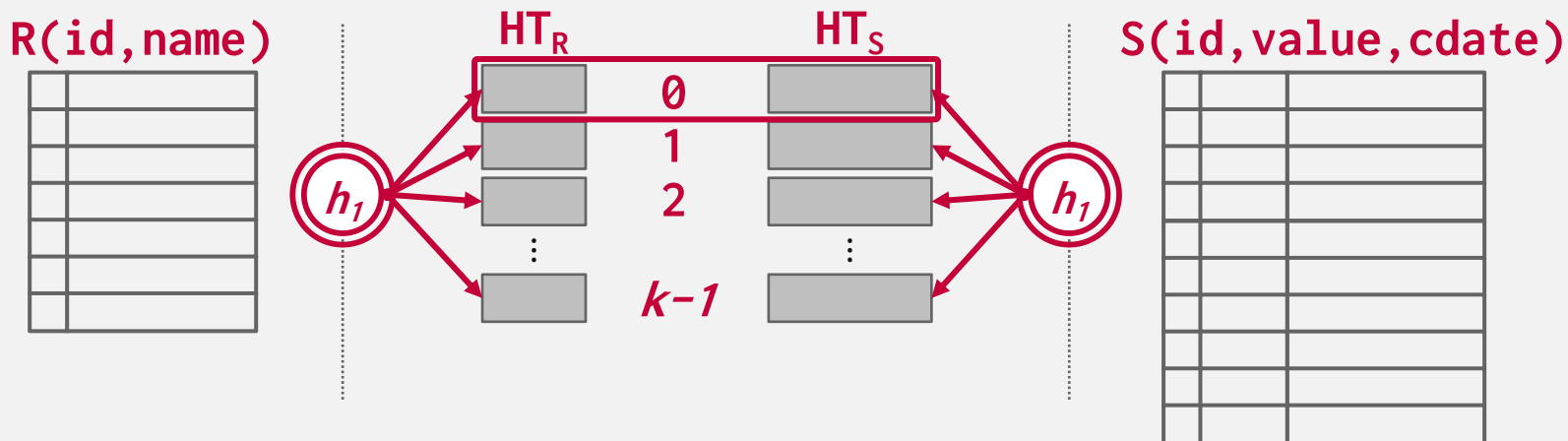
Hash **S** into k buckets with same hash function.

Write buckets to disk when they get full.



PARTITIONED HASH JOIN PROBE PHASE

Read corresponding partitions into memory one pair at a time, hash join their contents.



PARTITIONED HASH JOIN EDGE CASES

If a partition does not fit in memory, recursively partition it with a different hash function

→ Repeat as needed

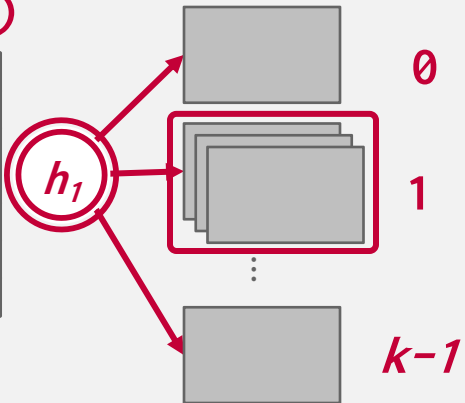
→ Eventually hash join the corresponding (sub-)partitions

If a single join key has so many matching records that they don't fit in memory, use a block nested loop join for that key

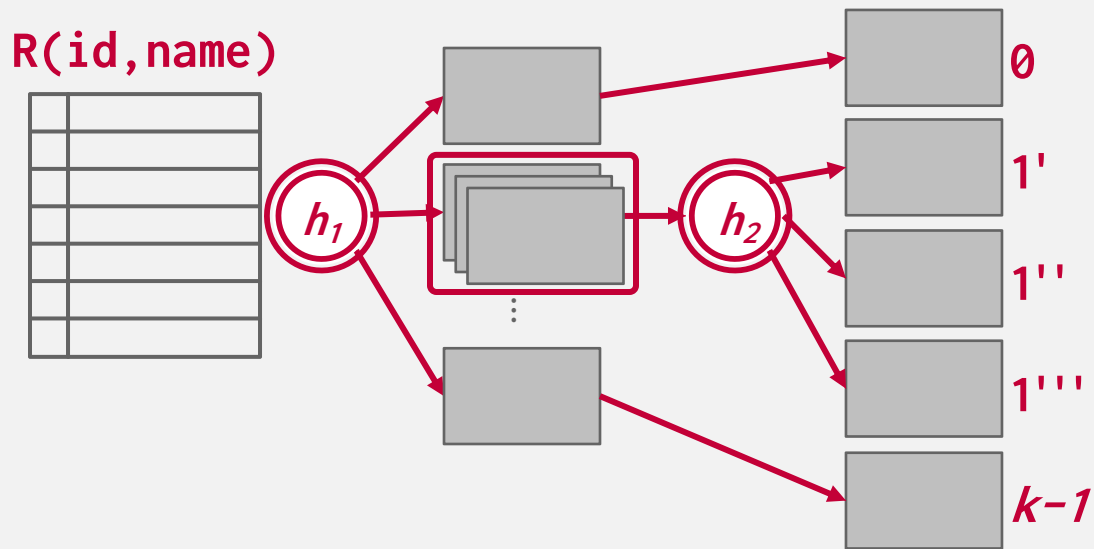
RECURSIVE PARTITIONING

$R(\text{id}, \text{name})$

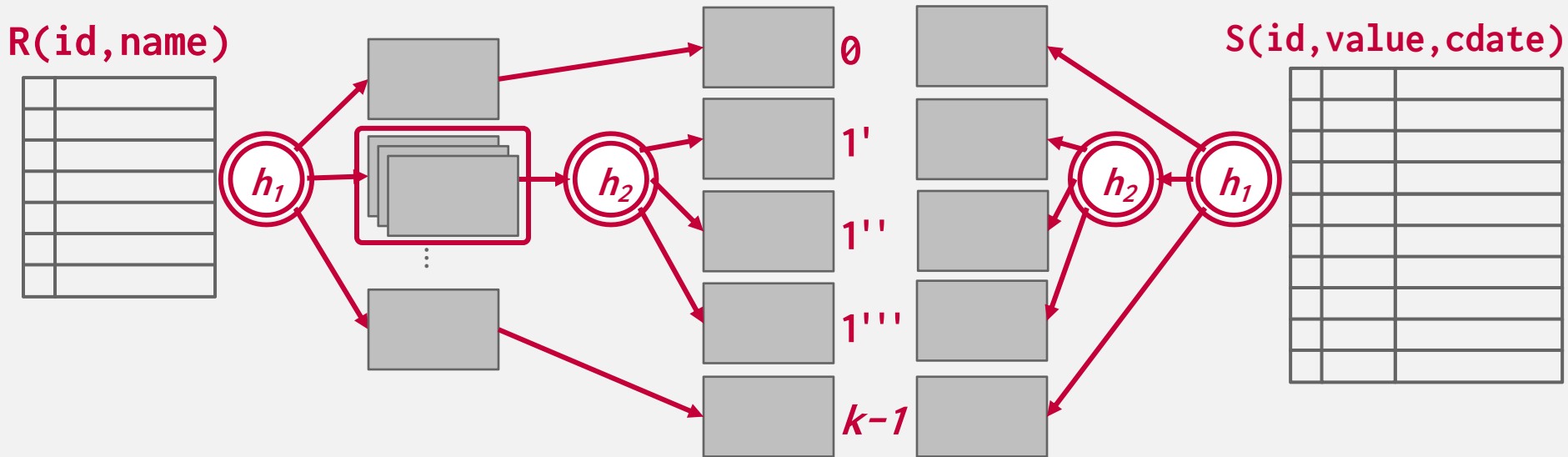
| | |
|--|--|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |



RECURSIVE PARTITIONING



RECURSIVE PARTITIONING



COST OF PARTITIONED HASH JOIN

If we do not need recursive partitioning:

→ Cost: $3(M + N)$

Partition phase:

→ Read+write both tables

→ $2(M+N)$ I/Os

Probe phase:

→ Read both tables (in total, one partition at a time)

→ $M+N$ I/Os

PARTITIONED HASH JOIN

Example database:

→ $M = 1000$, $m = 100,000$

→ $N = 500$, $n = 40,000$

Cost Analysis:

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

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

OPTIMIZATION: HYBRID HASH JOIN

During the partitioning phase, keep Partition 0 for R (aka. R_0) in memory.

- e.g. $B = 1000$ and $R = 5000$. Only need to partition R 5-ways (so each partition can fit in memory for the second phase). Keep R_0 in memory, in a hash table.
- In the second phase, for tuples that map to S_0 , simply probe the R_0 hash table.
- If the keys are skewed, could adjust the hash function to map the hot keys to R_0 .
- Difficult to get to work correctly. Rarely done in practice. Better to “over partition.”

R(id, name)

| | |
|--|--|
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |



0
1
2
⋮
max



S(id, value, cdate)

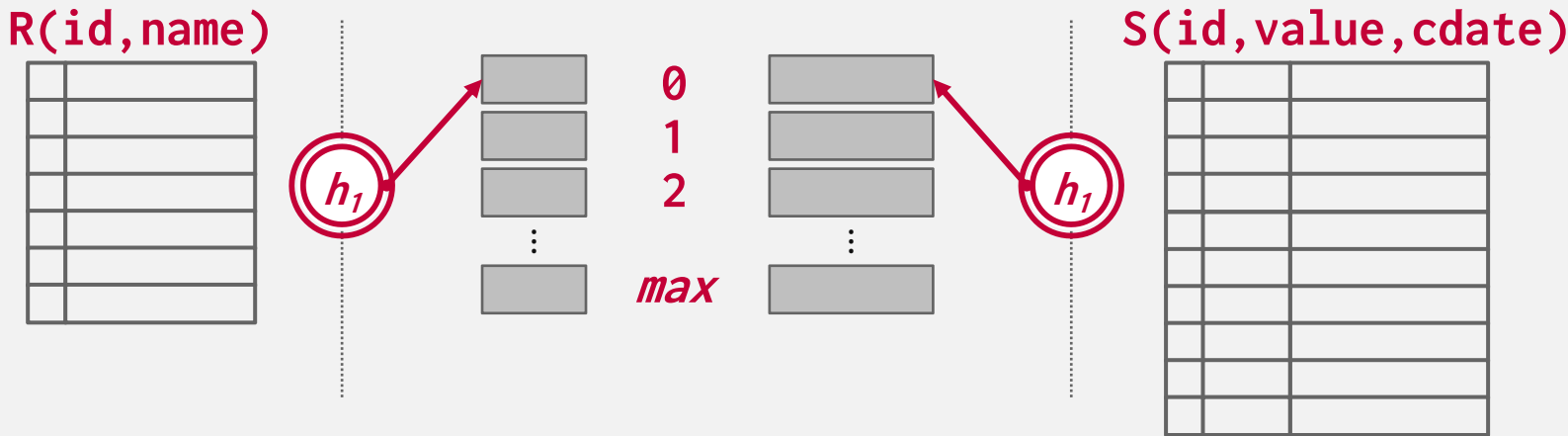
| | | |
|--|--|--|
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |
| | | |



OPTIMIZATION: HYBRID HASH JOIN

During the partitioning phase, keep Partition 0 for R (aka. R_0) in memory.

- e.g. $B = 1000$ and $R = 5000$. Only need to partition R 5-ways (so each partition can fit in memory for the second phase). Keep R_0 in memory, in a hash table.
- In the second phase, for tuples that map to S_0 , simply probe the R_0 hash table.
- If the keys are skewed, could adjust the hash function to map the hot keys to R_0 .
- Difficult to get to work correctly. Rarely done in practice. Better to “over partition.”



OPTIMIZATION: HYBRID HASH JOIN

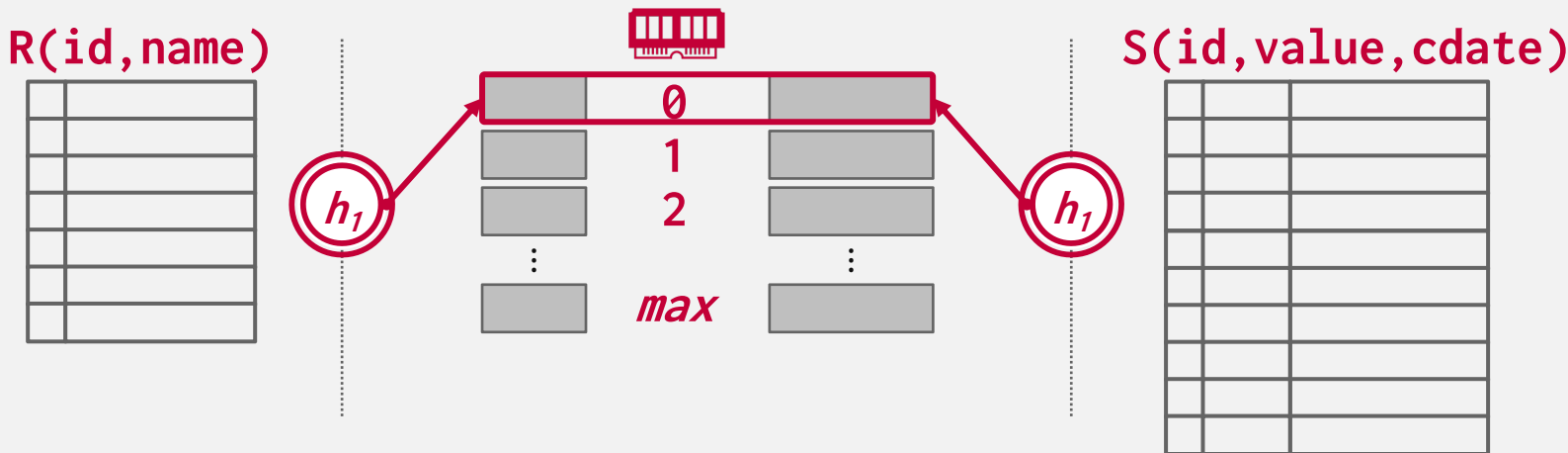
During the partitioning phase, keep Partition 0 for R (aka. R_0) in memory.

→ e.g. $B = 1000$ and $R = 5000$. Only need to partition R 5-ways (so each partition can fit in memory for the second phase). Keep R_0 in memory, in a hash table.

→ In the second phase, for tuples that map to S_0 , simply probe the R_0 hash table.

→ If the keys are skewed, could adjust the hash function to map the hot keys to R_0 .

→ Difficult to get to work correctly. Rarely done in practice. Better to “over partition.”



OPTIMIZATION: HYBRID HASH JOIN

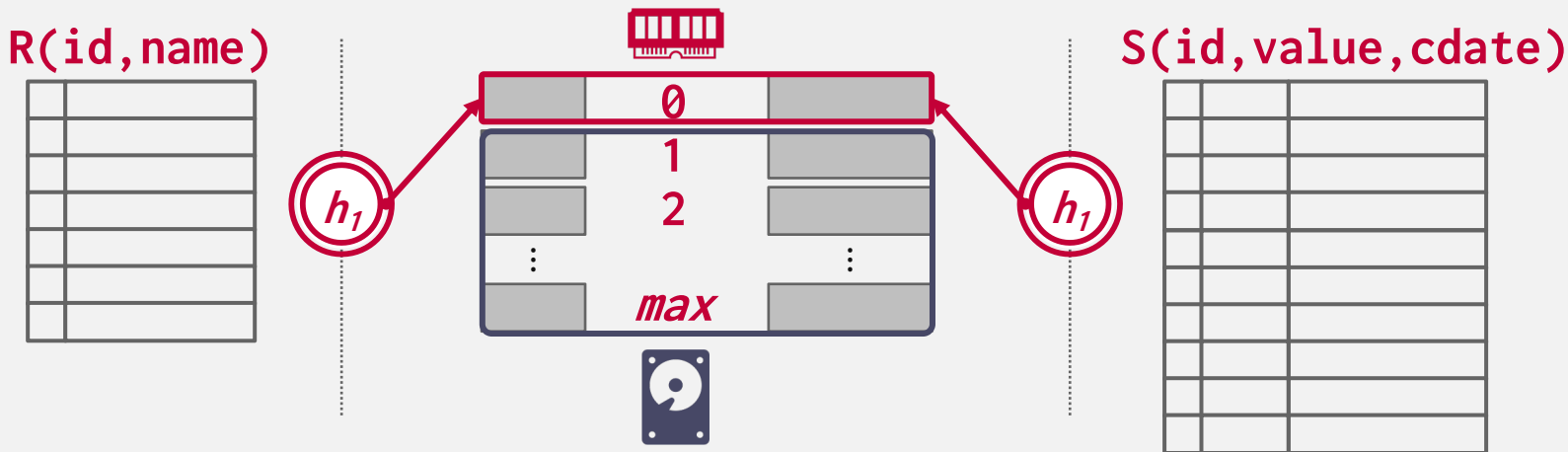
During the partitioning phase, keep Partition 0 for R (aka. R_0) in memory.

→ e.g. $B = 1000$ and $R = 5000$. Only need to partition R 5-ways (so each partition can fit in memory for the second phase). Keep R_0 in memory, in a hash table.

→ In the second phase, for tuples that map to S_0 , simply probe the R_0 hash table.

→ If the keys are skewed, could adjust the hash function to map the hot keys to R_0 .

→ Difficult to get to work correctly. Rarely done in practice. Better to “over partition.”



HASH JOIN OBSERVATIONS

The probe-side table can be any size.

→ Only the build-side table (or its partitions) need to fit in memory

If we know the size of the build-side table, then we can use a static hash table.

→ Less computational overhead

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 |
|------------------------|---|--------------|
| Naïve Nested Loop Join | $M + (m \cdot N)$ | 1.3 hours |
| Block Nested Loop Join | $M + (\lceil M / (B-2) \rceil \cdot N)$ | 0.55 seconds |
| Index Nested Loop Join | $M + (m \cdot C)$ | Variable |
| Sort-Merge Join | $M + N + (\text{sort cost})$ | 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.