

# 11

# Join Algorithms



Intro to Database Systems  
15-445/15-645  
Fall 2020

AP

Andy Pavlo  
Computer Science  
Carnegie Mellon University

# ADMINISTRIVIA

---

**Homework #2** is due Sunday Oct 4<sup>th</sup>

**Project #2** is now released:

- Checkpoint #1: Due Sunday Oct 11<sup>th</sup>
- Checkpoint #2: Due Sunday Oct 25<sup>th</sup>

**Mid-Term Exam** is Wed Oct 21<sup>st</sup>

- Download + Submit via Gradescope.
- We will offer two sessions based on your reported timezone in S3.



# WHY DO WE NEED TO JOIN?

---

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

We use the join operate 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 only in research literature.

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

# JOIN OPERATORS

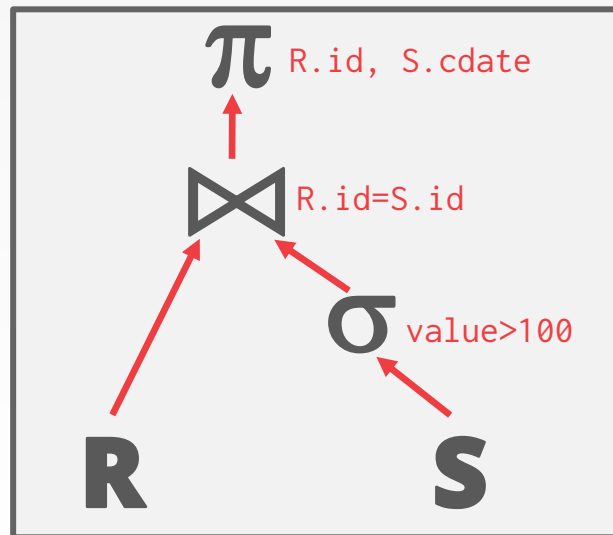
## Decision #1: Output

→ 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
```



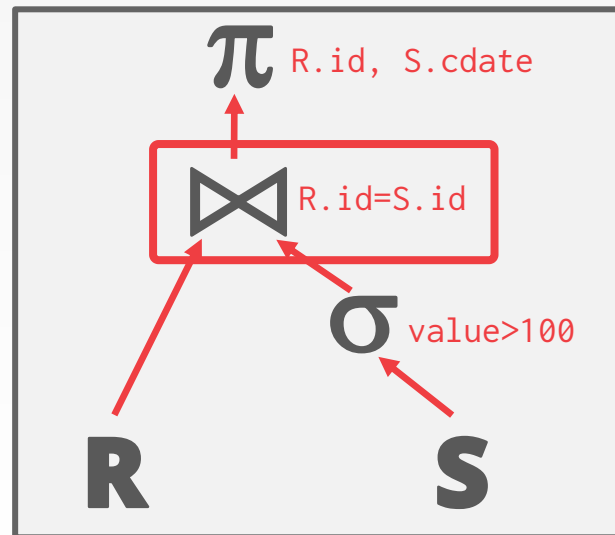
# OPERATOR OUTPUT

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	10/7/2020
			123	2000	10/7/2020

R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/7/2020
123	abc	123	2000	10/7/2020

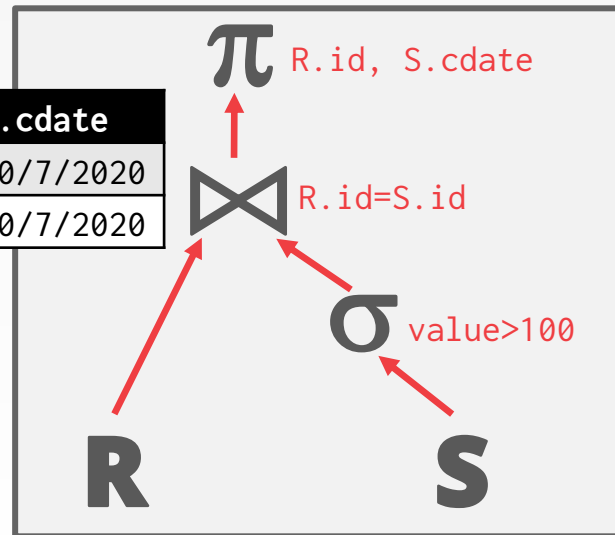
# OPERATOR OUTPUT: DATA

## 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	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/7/2020
123	abc	123	2000	10/7/2020





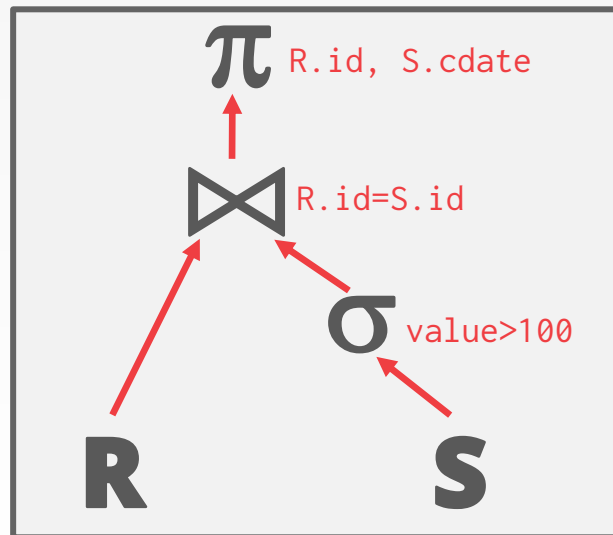
# OPERATOR OUTPUT: DATA

## 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	10/7/2020
			123	2000	10/7/2020

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

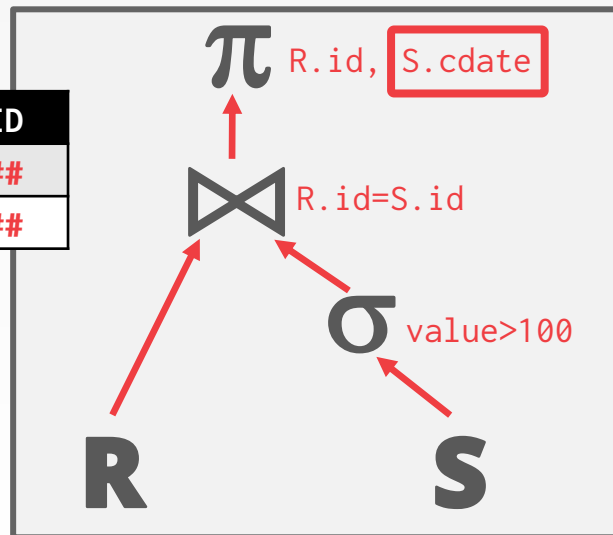
# OPERATOR OUTPUT: RECORD IDS

## 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.###



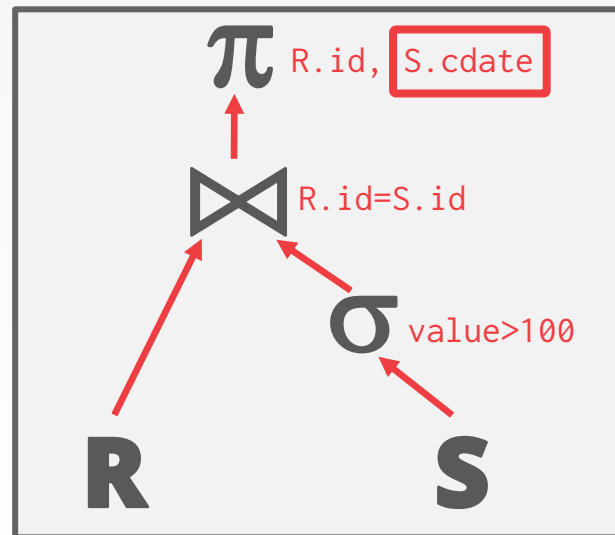
# OPERATOR OUTPUT: RECORD IDS

## 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 need 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 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

---

**$R \bowtie S$**  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

- Simple / Stupid
- Block
- Index

## Sort-Merge Join

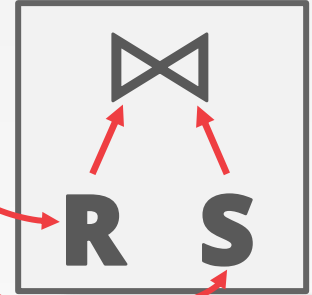
## Hash Join ■



# 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	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020



# STUPID NESTED LOOP JOIN

Why is this algorithm stupid?

→ 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	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

**$N$**  pages  
 **$n$**  tuples

# STUPID 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  $s \in B_S$ :
        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	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

$N$  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	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

**$N$**  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.

$M$  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	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

$N$  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  $\approx 50$  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	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

$N$  pages  
 $n$  tuples

# BLOCK NESTED LOOP JOIN

```

foreach  $B-2$  blocks  $b_R \in R$ :
  foreach block  $b_S \in S$ :
    foreach tuple  $r \in B-2$  blocks:
      foreach tuple  $s \in b_S$ :
        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	cdates
100	2222	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

$N$  pages  
 $n$  tuples



# BLOCK NESTED LOOP JOIN

---

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

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

■ 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  $\approx 0.15$  seconds



# NESTED LOOP JOIN

---

- Why does the basic nested loop join suck?
  - 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.
- Build one on the fly (hash table, B+Tree).



# INDEX NESTED LOOP JOIN

```

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

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

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

$M$  pages  
 $m$  tuples

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

id	value	cdate
100	2222	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

$N$  pages  
 $n$  tuples

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



# 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	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

**Index(S.id)**



$N$  pages  
 $n$  tuples

# NESTED LOOP JOIN

---

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.



# SORT-MERGE JOIN

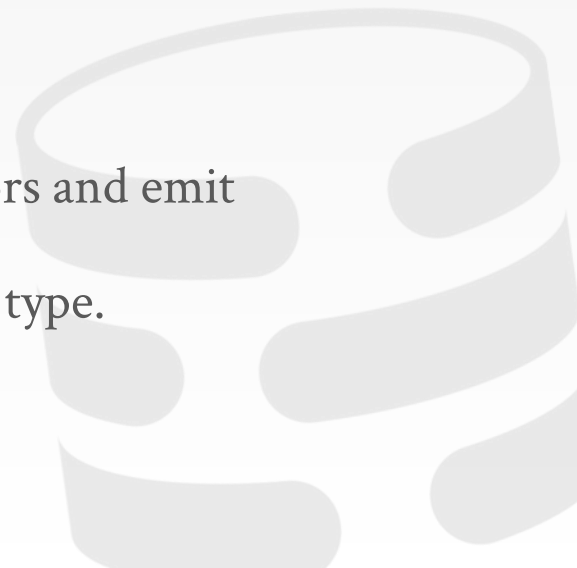
---

## 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
cursorR ← Rsorted, cursorS ← Ssorted
while cursorR and cursorS:
  if cursorR > cursorS:
    increment cursorS
  if cursorR < cursorS:
    increment cursorR
  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

↑  
**Sort!**

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

id	value	cdate
100	2222	10/7/2020
500	7777	10/7/2020
400	6666	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020

↑  
**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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

↑  
**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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020


```
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	10/7/2020


# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020


```
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	10/7/2020
100	Andy	100	9999	10/7/2020


# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020



# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020
400	Raekwon	200	6666	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020



```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020
400	Raekwon	200	6666	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020
400	Raekwon	200	6666	10/7/2020
500	RZA	500	7777	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020
400	Raekwon	200	6666	10/7/2020
500	RZA	500	7777	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020
400	Raekwon	200	6666	10/7/2020
500	RZA	500	7777	10/7/2020

# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020
400	Raekwon	200	6666	10/7/2020
500	RZA	500	7777	10/7/2020



# 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	10/7/2020
100	9999	10/7/2020
200	8888	10/7/2020
400	6666	10/7/2020
500	7777	10/7/2020

```
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	10/7/2020
100	Andy	100	9999	10/7/2020
200	GZA	200	8888	10/7/2020
200	GZA	200	8888	10/7/2020
400	Raekwon	200	6666	10/7/2020
500	RZA	500	7777	10/7/2020

# 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**) =  $2000 \cdot (1 + \lceil \log_{99} 1000 / 100 \rceil) = 4000$  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  $\approx 0.75$  seconds

# SORT-MERGE JOIN

---

The worst case for the merging phase is when the join attribute of all the tuples in both relations contain 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 by 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  $\mathbf{s}_i$ .

# BASIC HASH JOIN ALGORITHM

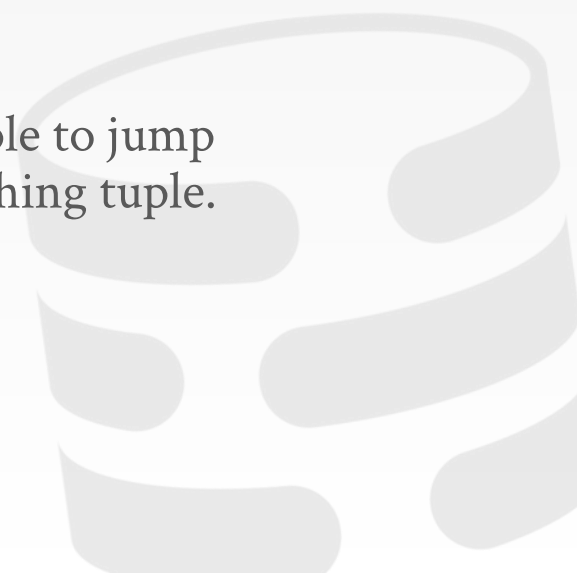
---

## Phase #1: Build

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

## Phase #2: Probe

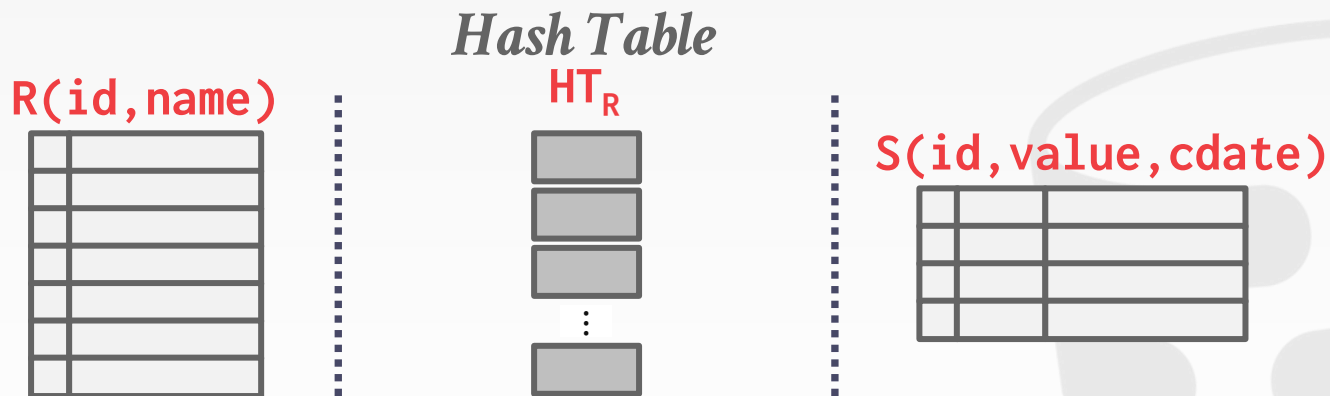
→ Scan the inner relation and use  $h_1$  on each tuple to jump to a location in the hash table and find a matching tuple.



# BASIC HASH JOIN ALGORITHM

```

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

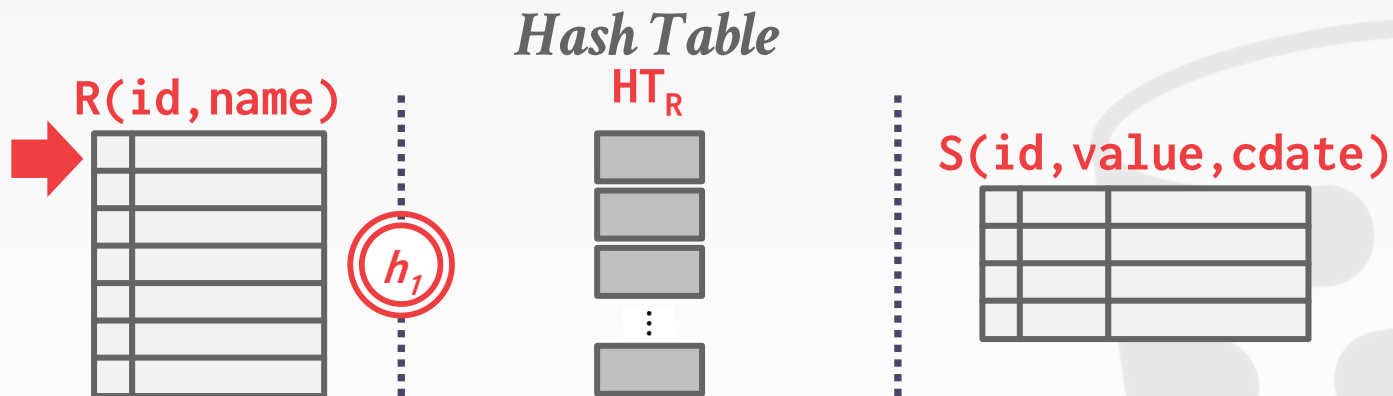




# BASIC HASH JOIN ALGORITHM

```

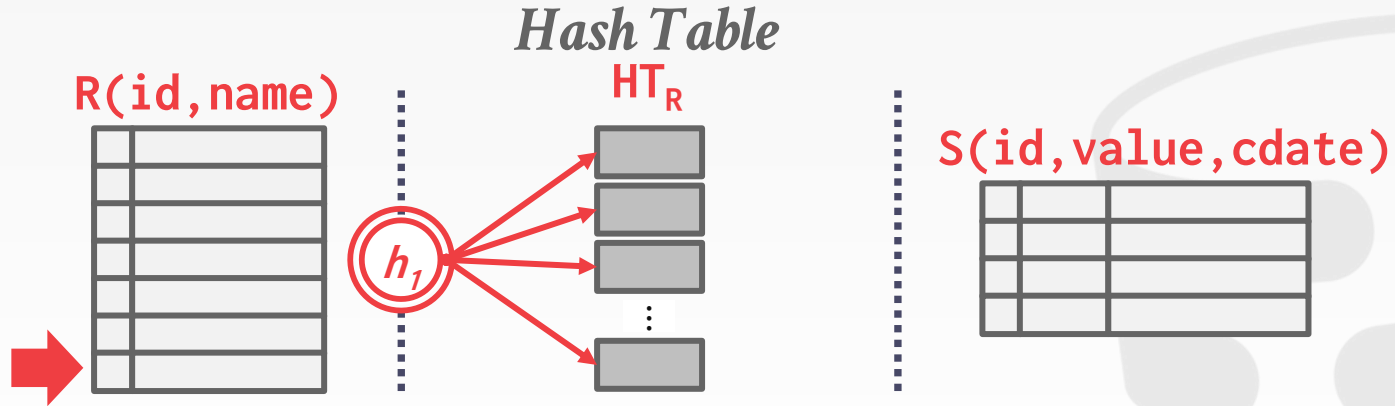
build hash table  $HT_R$  for  $R$ 
foreach tuple  $s \in S$ 
  output, if  $h_1(s) \in HT_R$ 
  
```



# BASIC HASH JOIN ALGORITHM

```

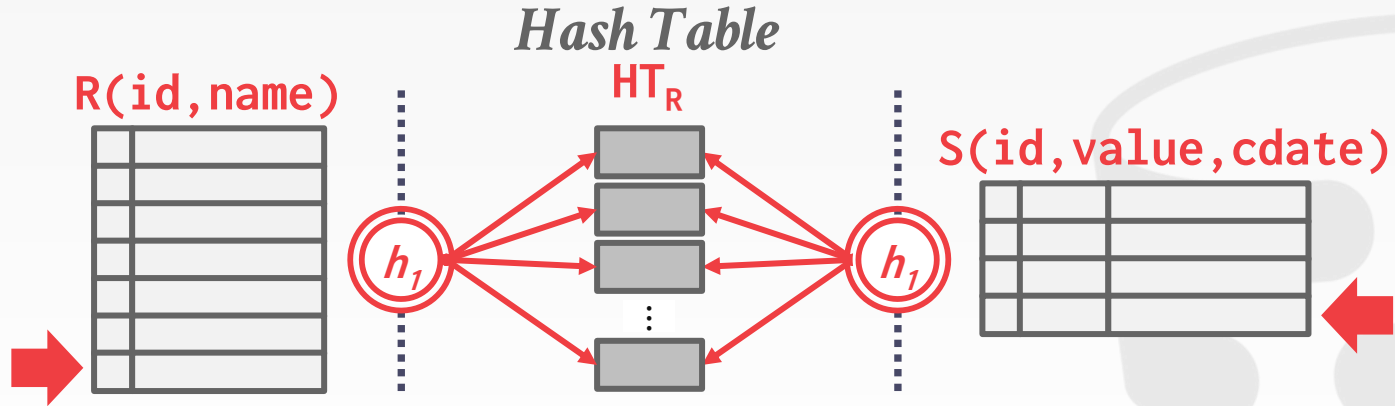
build hash table  $HT_R$  for  $R$ 
foreach tuple  $s \in S$ 
  output, if  $h_1(s) \in HT_R$ 
  
```



# BASIC HASH JOIN ALGORITHM

```

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



# HASH TABLE CONTENTS

---

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

**Value:** Varies per implementation.

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



# HASH TABLE VALUES

---

## **Approach #1: Full Tuple**

- Avoid having to retrieve the outer relation's tuple contents on a match.
- Takes up more space in memory.

## **Approach #2: Tuple Identifier**

- Could be to either the base tables or the intermediate output from child operators in the query plan.
- Ideal for column stores because the DBMS does not fetch data from disk it does not need.
- Also better if join selectivity is low.



# PROBE PHASE OPTIMIZATION

---

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

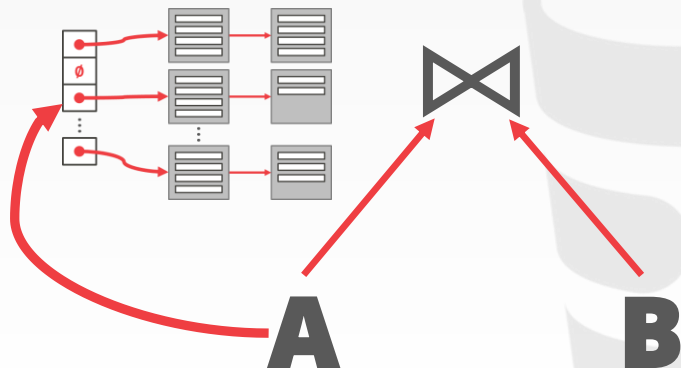
- Threads check the filter before probing the hash table.  
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



# PROBE PHASE OPTIMIZATION

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

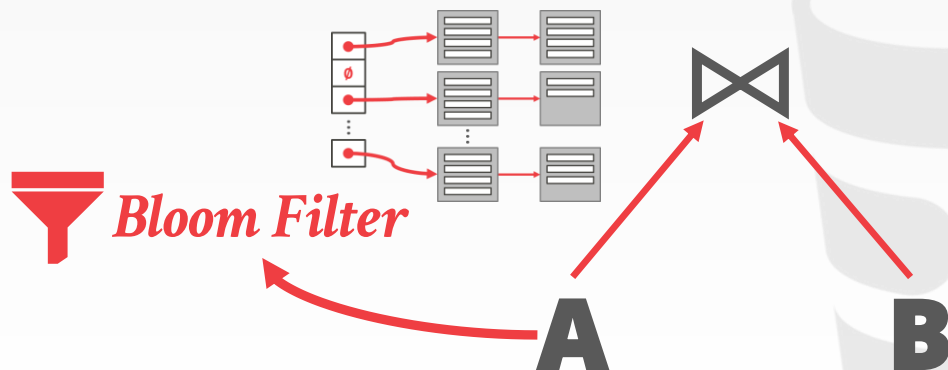
- Threads check the filter before probing the hash table.  
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



# PROBE PHASE OPTIMIZATION

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*.

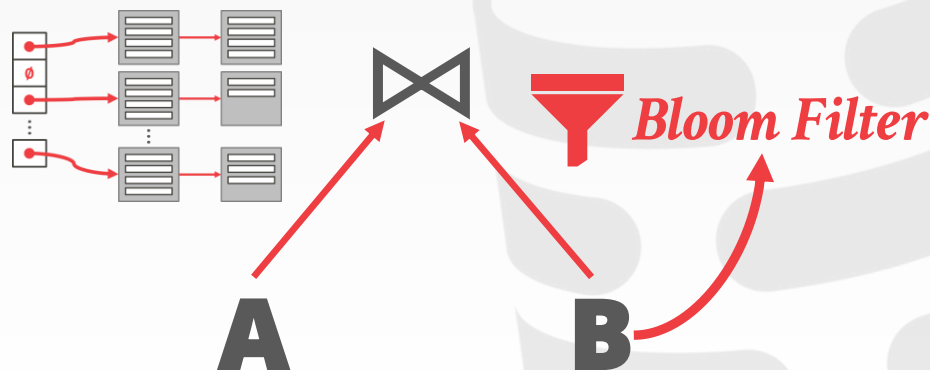




# PROBE PHASE OPTIMIZATION

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

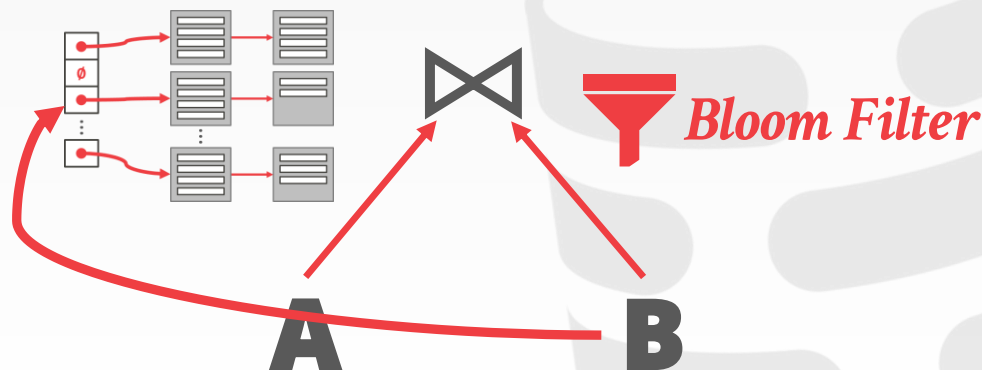
- Threads check the filter before probing the hash table.  
This will be faster since the filter will fit in CPU caches.
- Sometimes called *sideways information passing*.



# PROBE PHASE OPTIMIZATION

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*.



# COST ANALYSIS

---

How big of a table can we hash using this approach?

- $B-1$  "spill partitions" in Phase #1
- Each should be no more than  $B$  blocks big

Answer:  $B \cdot (B-1)$

- A table of  $N$  pages needs about  $\text{sqrt}(N)$  buffers
- Assumes hash distributes records evenly.  
Use a "fudge factor"  $f > 1$  for that: we need  $B \cdot \text{sqrt}(f \cdot N)$



# 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 a random.



# GRACE 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.

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



GRACE  
University of Tokyo

# GRACE HASH JOIN

---

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

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

**R(id, name)**



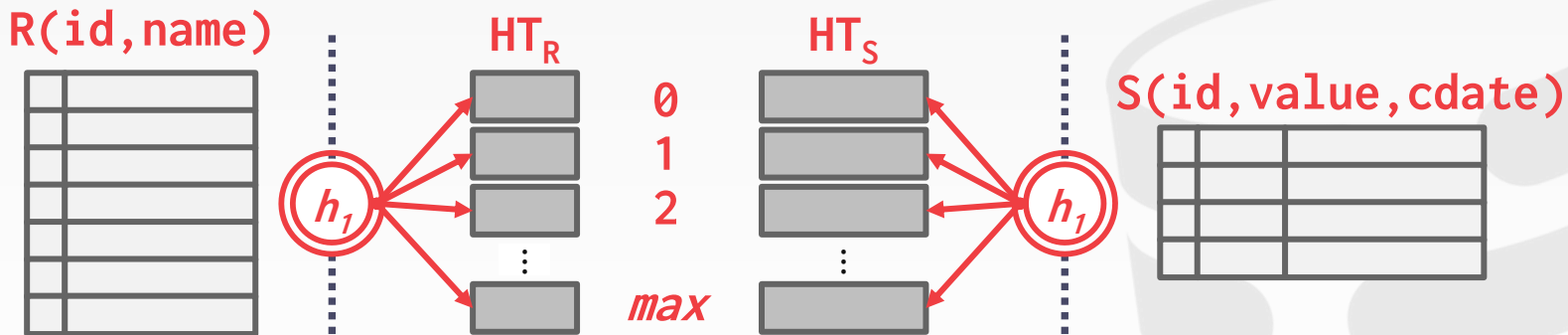

**S(id, value, cdate)**




# GRACE HASH JOIN

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

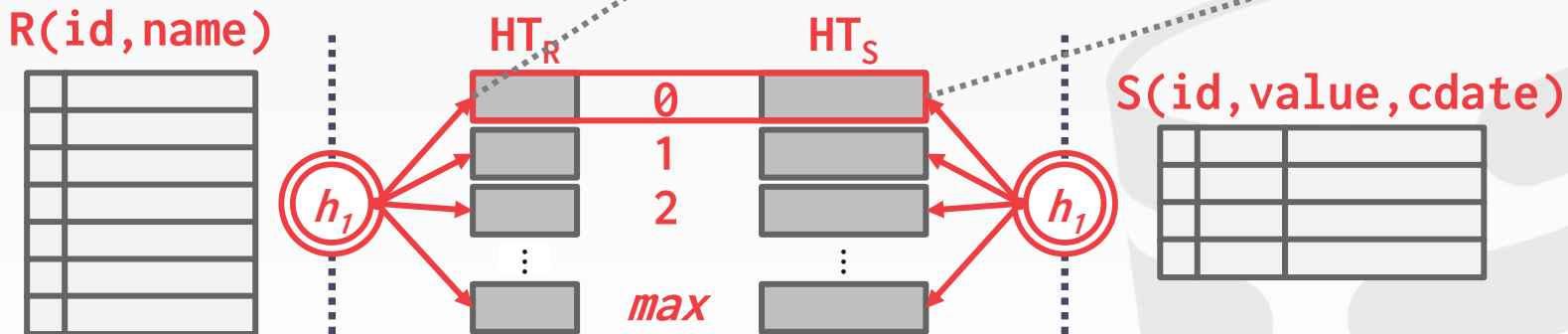
Hash **S** into the same # of buckets with the same hash function.



# GRACE HASH JOIN

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

```
foreach tuple  $r \in \text{bucket}_{R,\theta}$ :
  foreach tuple  $s \in \text{bucket}_{S,\theta}$ :
    emit, if match( $r, s$ )
```





# GRACE HASH JOIN

---

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

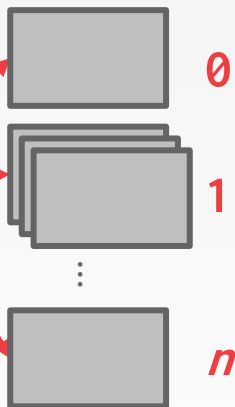
- Build another hash table for **bucket<sub>R,i</sub>** using hash function  **$h_2$**  (with  **$h_2 \neq h_1$** ).
- Then probe it for each tuple of the other table's bucket at that level.



# RECURSIVE PARTITIONING

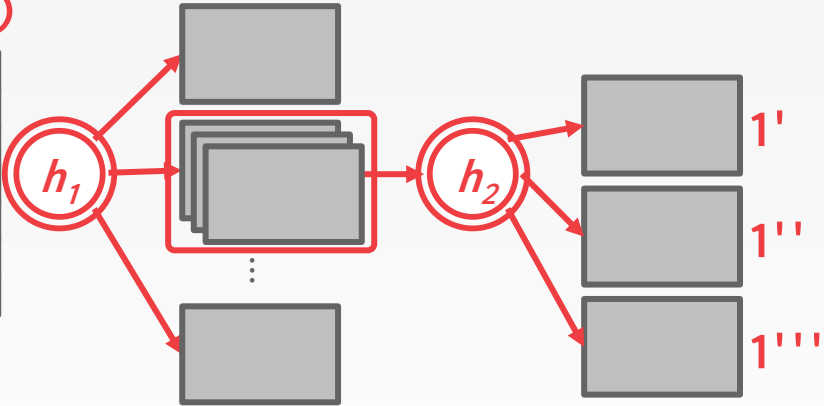
---

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

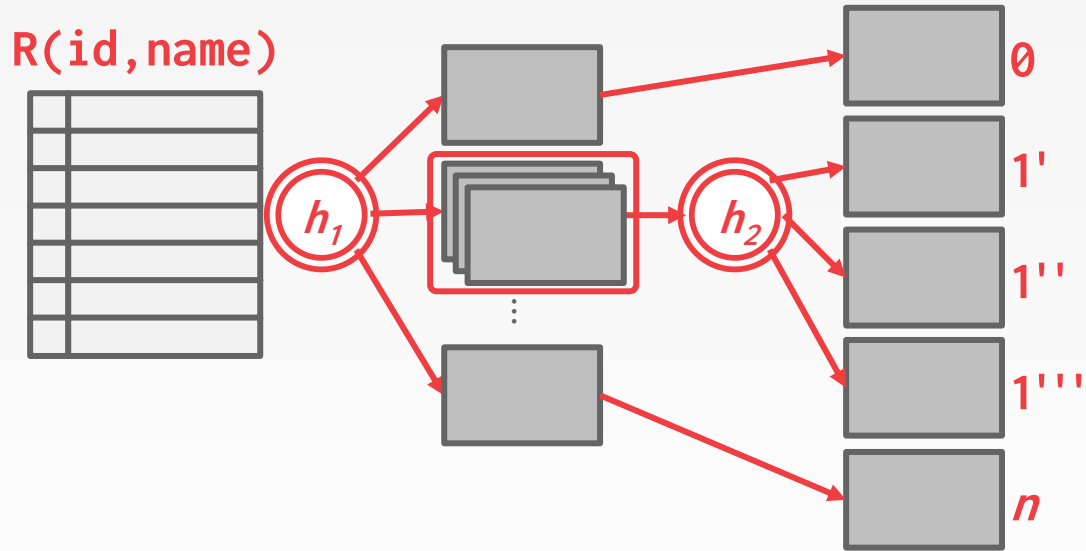



# RECURSIVE PARTITIONING

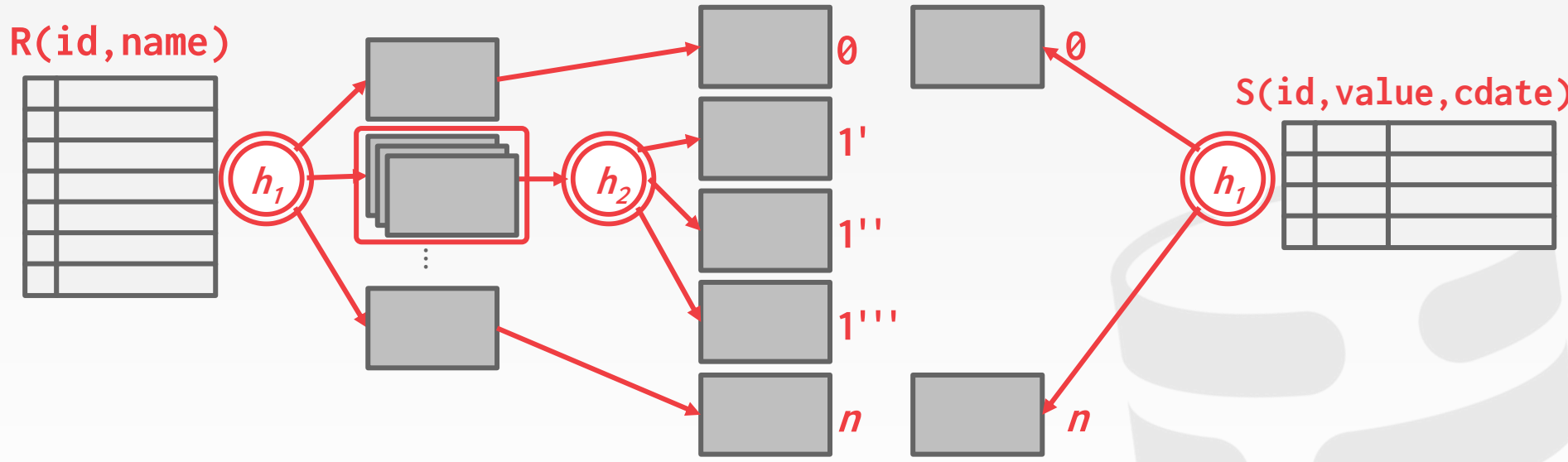
$R(\text{id}, \text{name})$

# RECURSIVE PARTITIONING



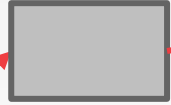
# RECURSIVE PARTITIONING



# RECURSIVE PARTITIONING

$R(id, name)$

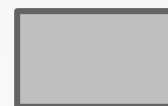

$h_1$



⋮



$h_2$

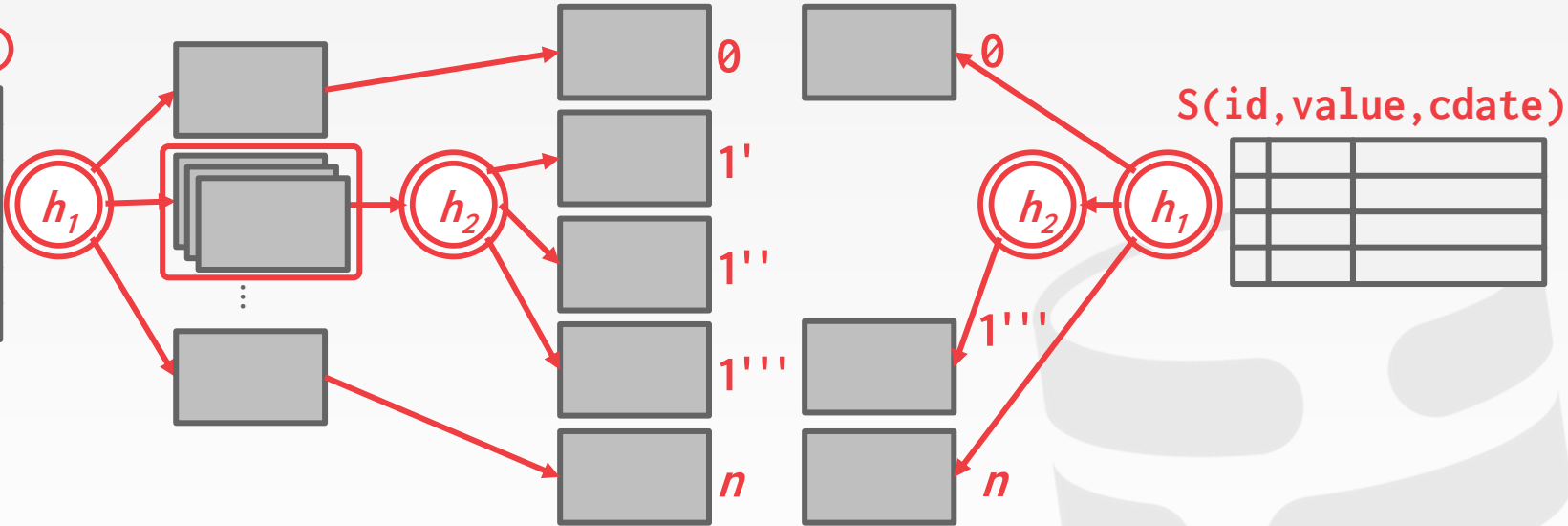


$S(id, value, cdate)$


$h_1$

# RECURSIVE PARTITIONING

$R(id, name)$

# GRACE 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





# GRACE 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  $\approx 0.45$  seconds



# OBSERVATION

---

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	$M + N + (\text{sort cost})$	0.75 seconds
Hash Join	$3(M + N)$	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.

