Carnegie Mellon University

Join Algorithms

Lecture #12

Database Systems
15-445/15-645
Fall 2018

AP
Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #2 – Checkpoint #1 is due TODAY

No class on Wednesday October 10th

Mid-term Exam is on Wednesday October 17th
→ Will cover up to and including this lecture (L12).
→ Study guide will be posted on Piazza later this week.
→ One sheet of handwritten notes (double-sided).

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.

NORMALIZED TABLES

ArtistAlbum(artist_id, album_id)

artist_id	album_id
123	11
123	22
789	22
456	22

Artist(id, name, year, country)

id	name	year	country
123	Wu Tang Clan	1992	USA
456	Notorious BIG	1992	USA
789	Ice Cube	1989	USA

Album(id, name, year)

id	name	year
11	Enter the Wu Tang	1993
22	St.Ides Mix Tape	1994
33	AmeriKKKa's Most Wanted	1990

JOIN ALGORITHMS

We will focus on joining **two** tables at a time.

In general, we want the smaller table to always be the outer table.

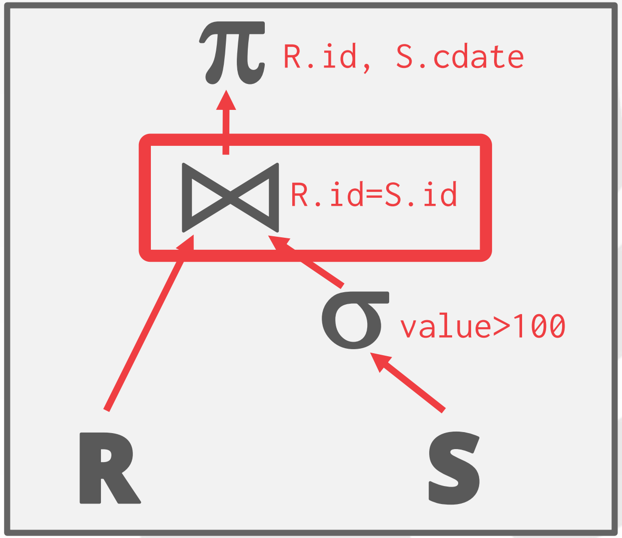
Things we need to discuss first:
→ Output
→ Cost Analysis Criteria

JOIN OPERATOR OUTPUT

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

Contents can vary:
→ Depends on processing model
→ Depends on storage model
→ Depends on the query

```sql
SELECT R.id, S.cdate
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```



JOIN OPERATOR OUTPUT: DATA

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

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

R(id,name)

id	name
123	abc

$\bowtie$

S(id,value,cdate)

id	value	cdate
123	1000	10/9/2018
123	2000	10/9/2018

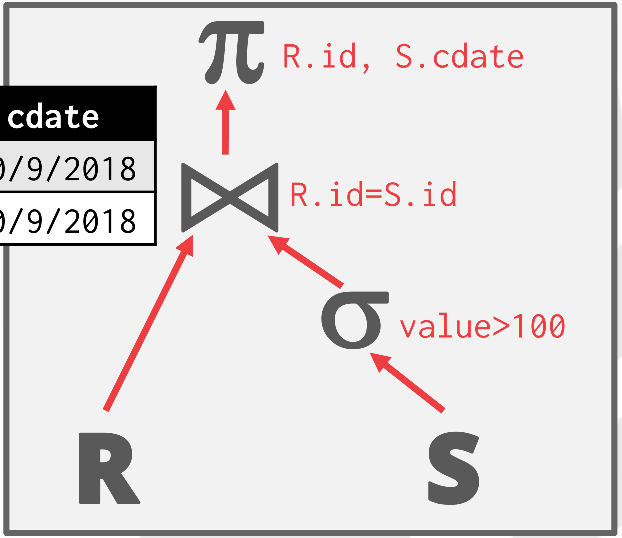
R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/9/2018
123	abc	123	2000	10/9/2018

JOIN OPERATOR OUTPUT: DATA

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

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

R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	10/9/2018
123	abc	123	2000	10/9/2018

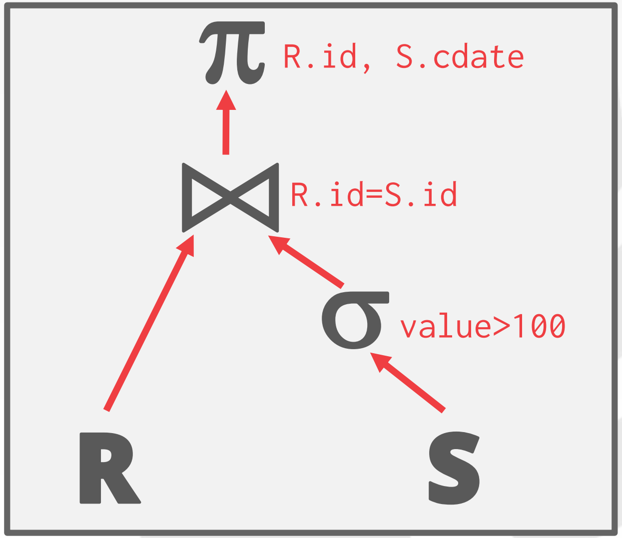


JOIN OPERATOR OUTPUT: DATA

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, S
 WHERE R.id = S.id
   AND S.value > 100
```



JOIN OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

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

A(id,name)

id	name
123	abc

⋈

S(id,value,cdate)

id	value	cdate
123	1000	10/9/2018
123	2000	10/9/2018

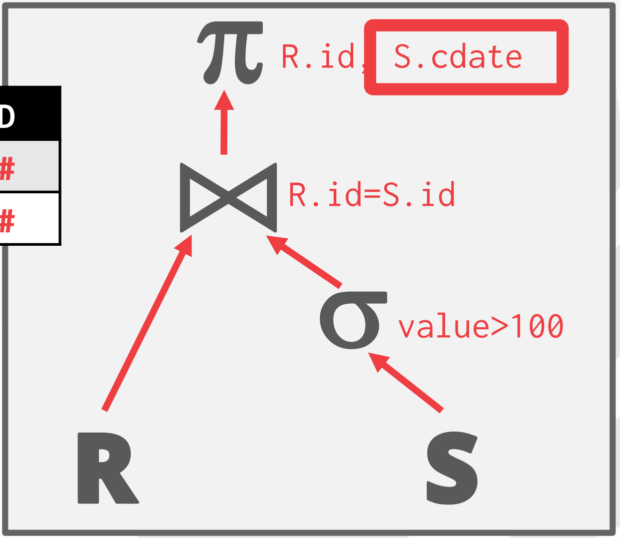
R.id	R.RID	S.id	S.RID
123	R.###	123	S.###
123	R.###	123	S.###

JOIN OPERATOR OUTPUT: RECORD IDS

Only copy the joins keys along with the record ids of the matching tuples.

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

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



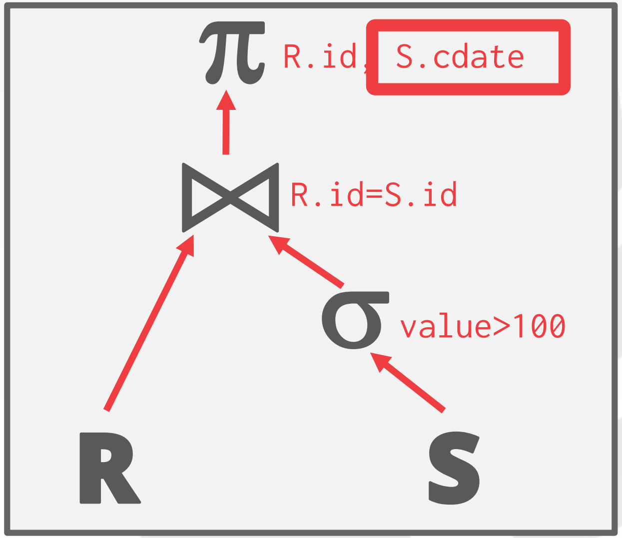
JOIN OPERATOR OUTPUT: RECORD IDS

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.

This is called **late materialization**.

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



I/O COST ANALYSIS

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

Assume:

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

→ N pages in **S**, n tuples total

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 particular algorithm works well in all scenarios.

JOIN ALGORITHMS

Nested Loop Join
→ Simple
→ Block
→ Index

Sort-Merge Join

Hash Join

SIMPLE 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	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

SIMPLE NESTED LOOP JOIN

Why is this algorithm bad?

→ For every tuple in **R**, it scans **S** once

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

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	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

N pages
n tuples

SIMPLE NESTED LOOP JOIN

Example database:

→ $M = 1000$, $m = 100{,}000$

→ $N = 500$, $n = 40{,}000$

Cost Analysis:

→ $M + (m \cdot N) = 1000 + (100000 \cdot 500) =$ **50,000,100 IOs**

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

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

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

→ At 0.1 ms/IO, Total time ≈ 1.1 hours

BLOCK NESTED LOOP JOIN

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

R(id,name)

M pages
m tuples

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	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

N pages
n tuples

BLOCK NESTED LOOP JOIN

This algorithm performs fewer disk accesses.

→ For every block in **R**, it scans **S** once

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

R(id,name)

M pages
m tuples

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

S(id,value,cdate)

N pages
n tuples

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

BLOCK NESTED LOOP JOIN

Which one should be the outer table?
→ The smaller table in terms of # of pages

R(id,name)

M pages
m tuples

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

S(id,value,cdate)

N pages
n tuples

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

BLOCK NESTED LOOP JOIN

Example database:

→ M = 1000, m = 100,000

→ N = 500, n = 40,000

Cost Analysis:

→ $M + (M \cdot N)$ = 1000 + (1000 · 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.

R(id,name)

M pages
m tuples

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

S(id,value,cdate)

N pages
n tuples

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

BLOCK NESTED LOOP JOIN

```
foreach B - 2 blocks bR ∈ R:
  foreach block bS ∈ S:
    foreach tuple r ∈ bR:
      foreach tuple s ∈ bS:
        emit, if r and s match
```

R(id,name)

M pages
m tuples

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

S(id,value,cdate)

N pages
n tuples

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

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

INDEX NESTED LOOP JOIN

Why do basic nested loop joins suck ass?

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

Can we accelerate the join using an index?

Use an index to find inner table matches.

→ We could use an existing index for the join.

→ Or even build one on the fly.

INDEX NESTED LOOP JOIN

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

R(id,name)

M pages
m tuples

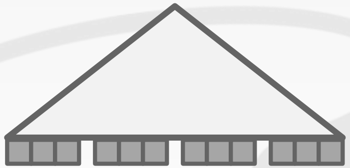
id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

N pages
n tuples

Index(S.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)`

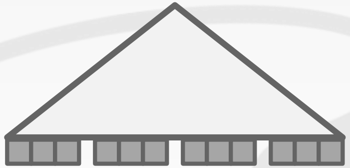
id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

`S(id,value,cdate)`

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

N pages
n tuples

`Index(S.id)`



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).
→ Can use the external merge sort algorithm that we talked about last class.

Phase #2: Merge

→ Step through the two sorted tables in parallel, 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
400	Raekwon

Sort!

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

Sort!

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

SORT-MERGE JOIN

R(id,name)

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

Sort!

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

Sort!

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

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018
200	GZA	200	8888	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018
200	GZA	200	8888	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018
200	GZA	200	8888	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018
200	GZA	200	8888	10/9/2018
400	Raekwon	200	8888	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018
200	GZA	200	8888	10/9/2018
400	Raekwon	200	8888	10/9/2018

SORT-MERGE JOIN

R(id,name)

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

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

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

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018
200	GZA	200	8888	10/9/2018
400	Raekwon	200	8888	10/9/2018
500	RZA	500	7777	10/9/2018

SORT-MERGE JOIN

Sort Cost (R): $2M \cdot (\log M / \log B)$

Sort Cost (S): $2N \cdot (\log N / \log B)$

Merge Cost: $(M + N)$

Total Cost: Sort + Merge

SORT-MERGE JOIN

Example database:

→ M = 1000, m = 100,000

→ N = 500, n = 40,000

With 100 buffer pages, both **R** and **S** can be sorted in two passes:

→ Sort Cost (**R**) = 2000 · (log 1000 / log 100) = **3000 IOs**

→ Sort Cost (**S**) = 1000 · (log 500 / log 100) = **1350 IOs**

→ Merge Cost = (1000 + 500) = **1500 IOs**

→ Total Cost = 3000 + 1350 + 1500 = **5850 IOs**

→ At 0.1 ms/IO, Total time ≈ 0.59 seconds

SORT-MERGE JOIN

The worst case for the merging phase is when the join attribute of all of the tuples in both relations contain 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 by 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 value i, the R tuple has to 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

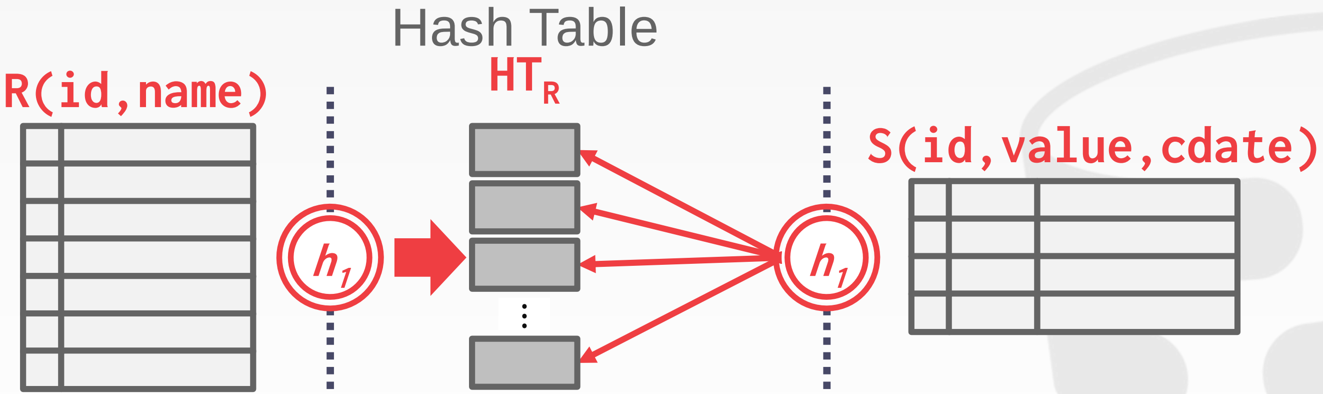
→ Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

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 ∈ S
  output, if h1(s) ∈ 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

→ Ideal for column stores because the DBMS doesn't fetch data from disk it doesn't need.

→ Also better if join selectivity is low.

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.

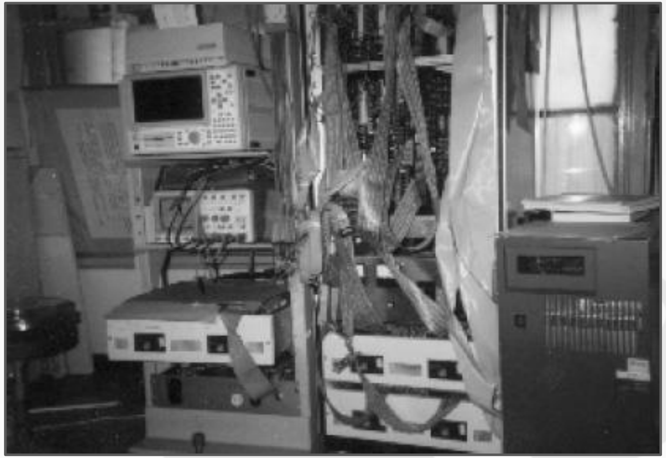
GRACE HASH JOIN

Hash join when tables don't fit in memory.

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

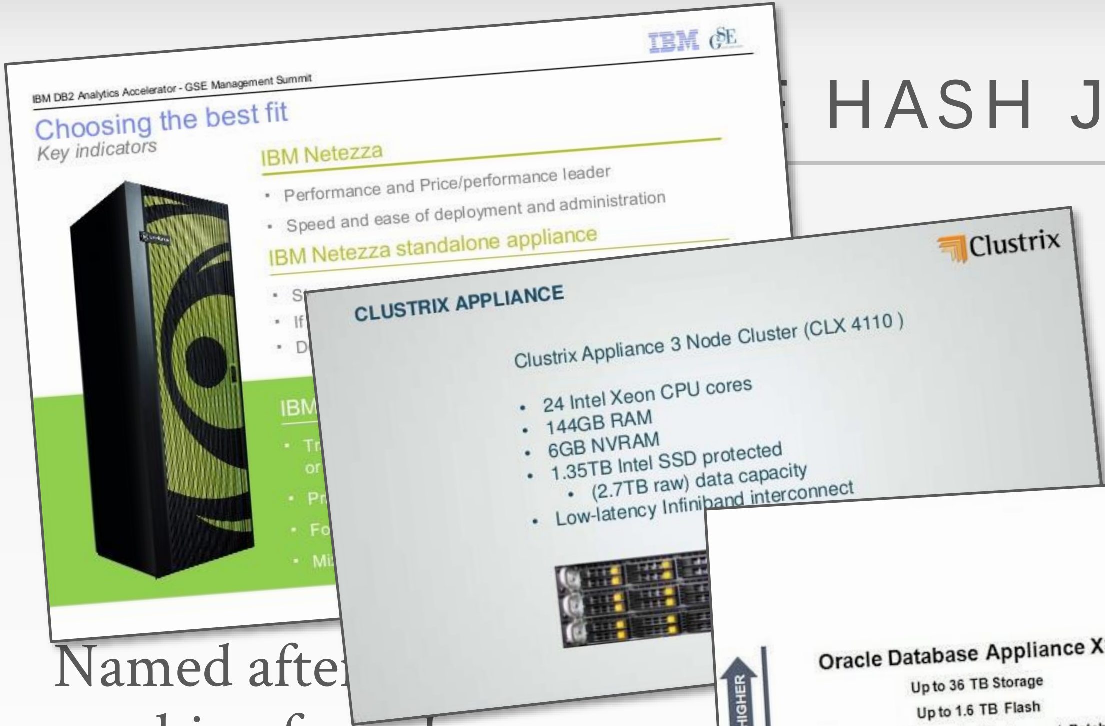
→ **Probe Phase:** Compares tuples in corresponding partitions for each table.

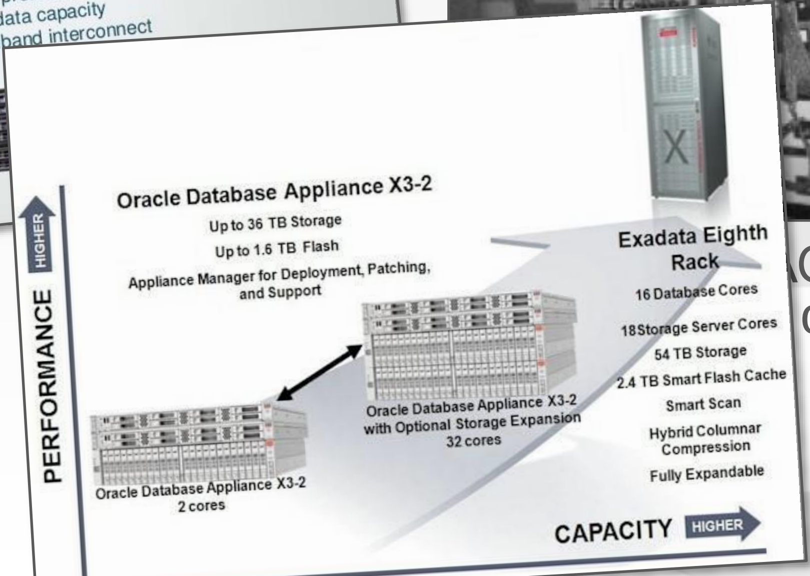
Named after the **GRACE** database machine from Japan.



GRACE
University of Tokyo

HASH JOIN





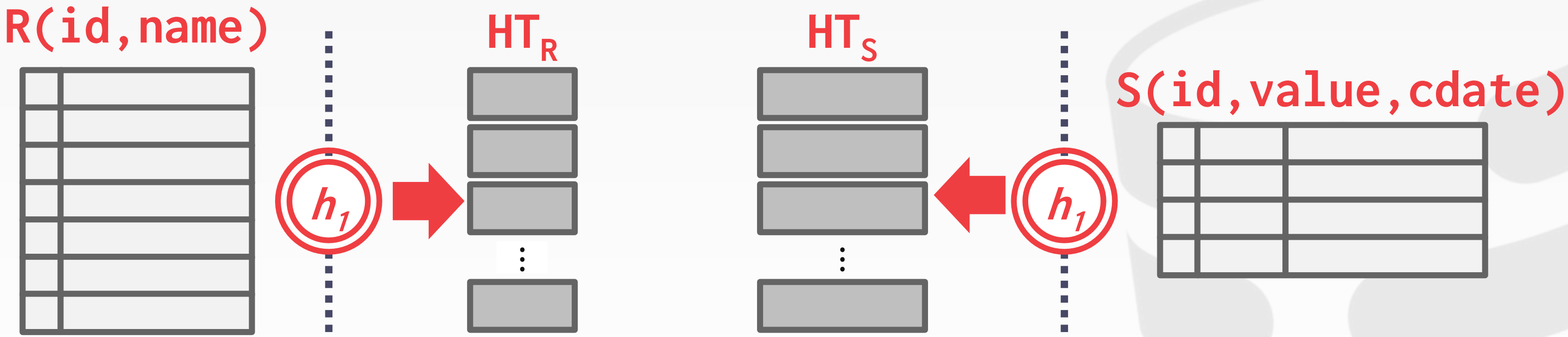
Named after … machine from Japan.

…ACE … of Tokyo

GRACE HASH JOIN

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

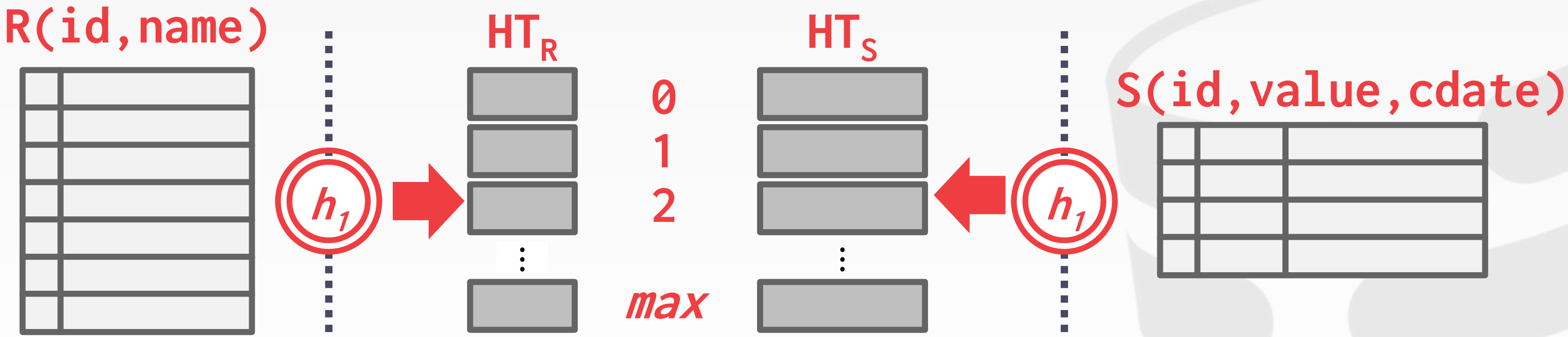
Hash **S** into the same # of buckets with the same hash function.



GRACE HASH JOIN

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

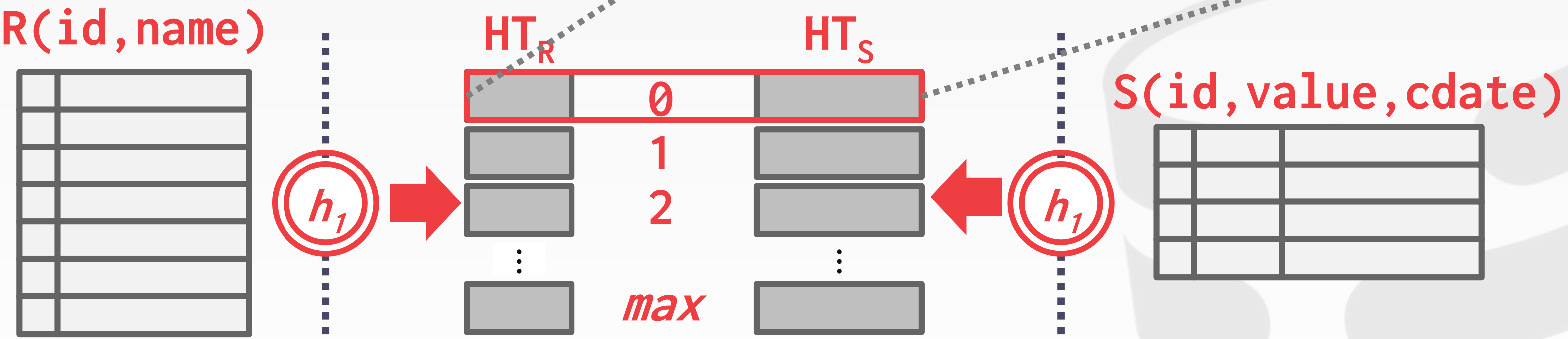
Hash **S** into the same # of buckets with the same hash function.



GRACE HASH JOIN

Join each pair of matching buckets between **R** and **S**.

```
foreach tuple r ∈ bucket_{R,0}:
  foreach tuple s ∈ bucket_{S,0}:
    emit, if match(r, s)
```



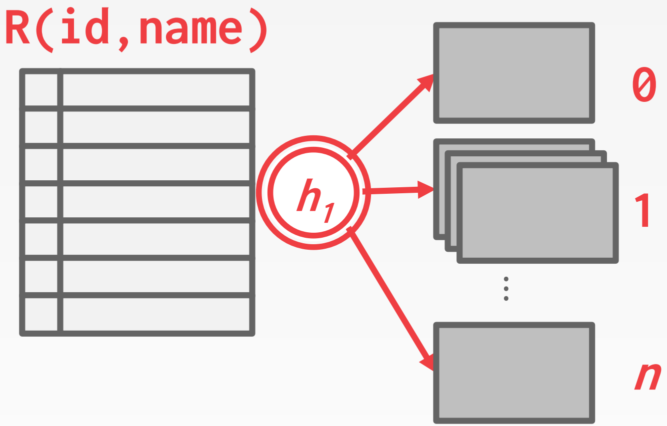
GRACE HASH JOIN

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

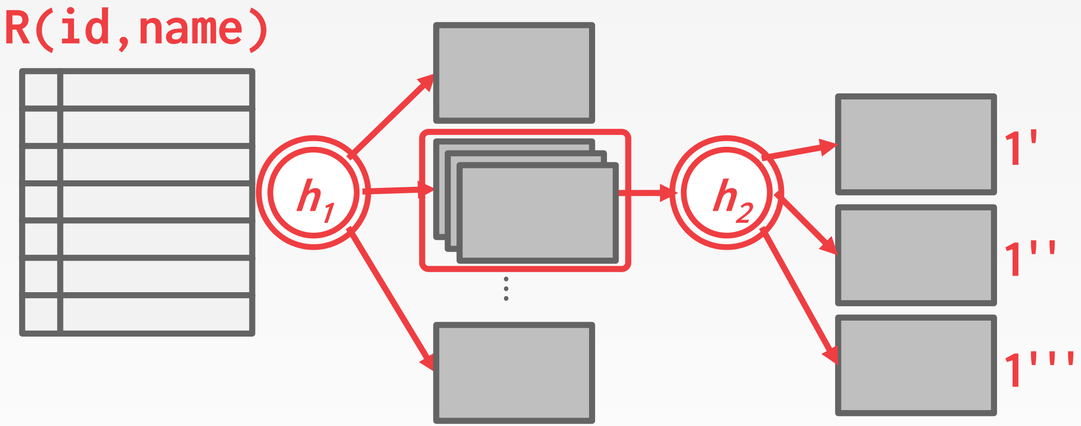
→ Build another hash table for **bucket$_{R,i}$** using hash function **h_2** (with **$h_2 \neq h_1$**).

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

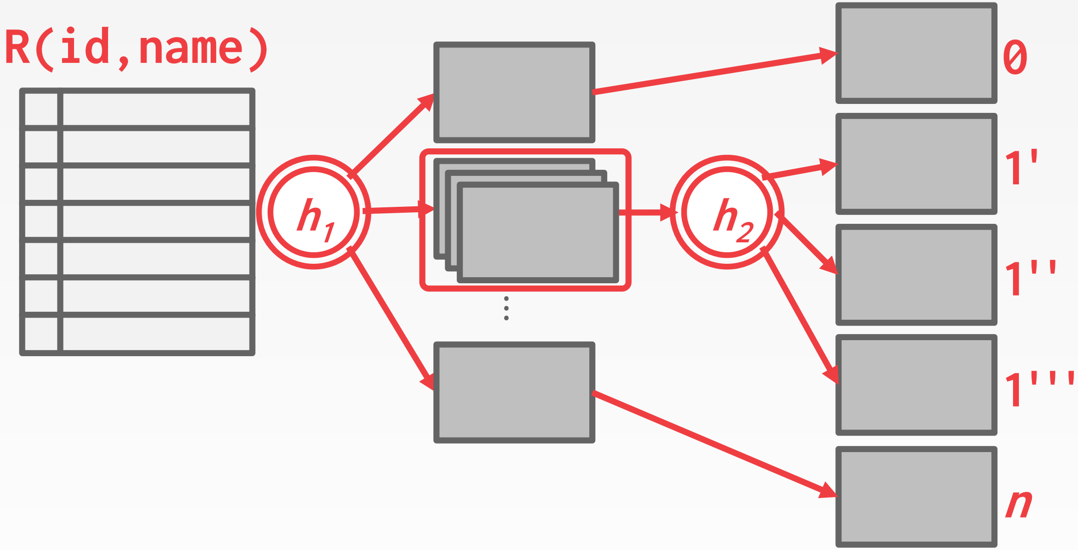
RECURSIVE PARTITIONING



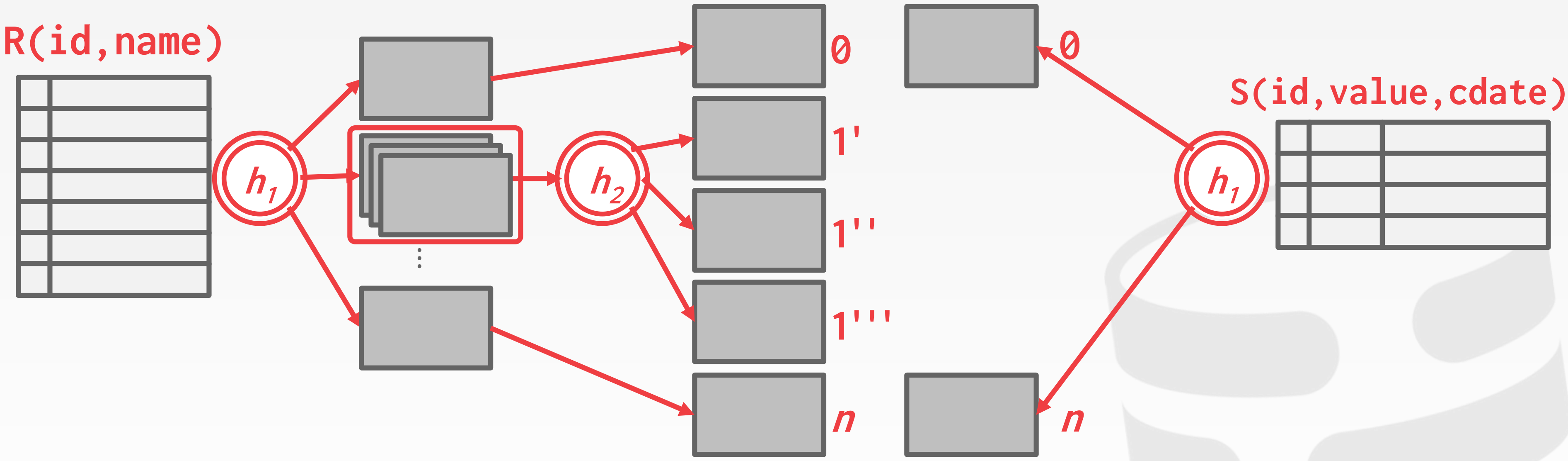
RECURSIVE PARTITIONING



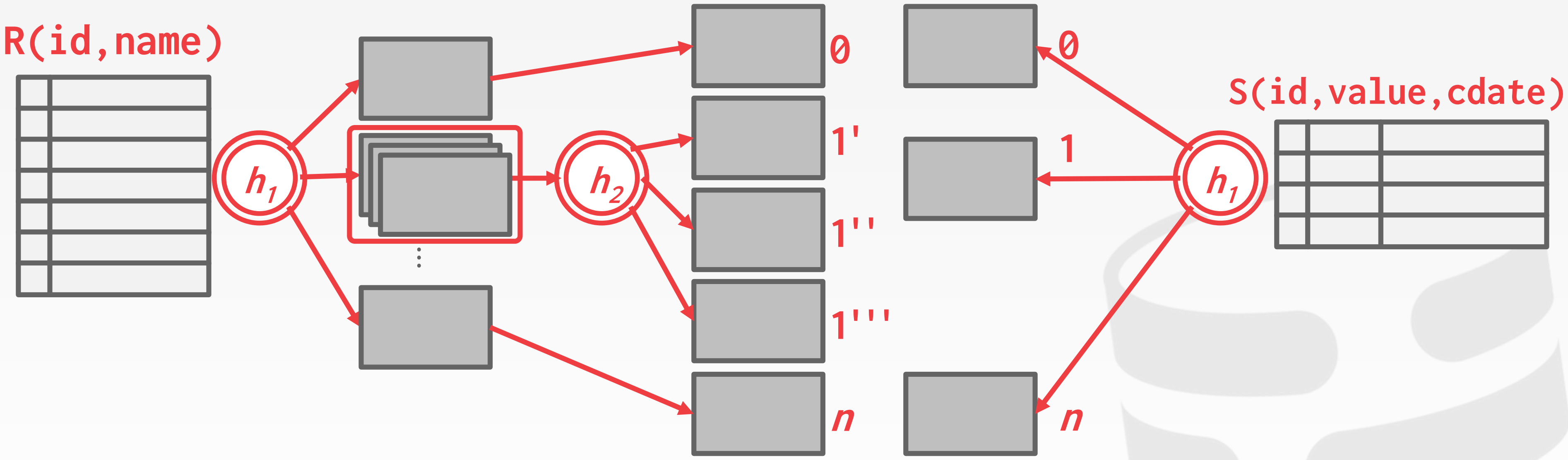
RECURSIVE PARTITIONING



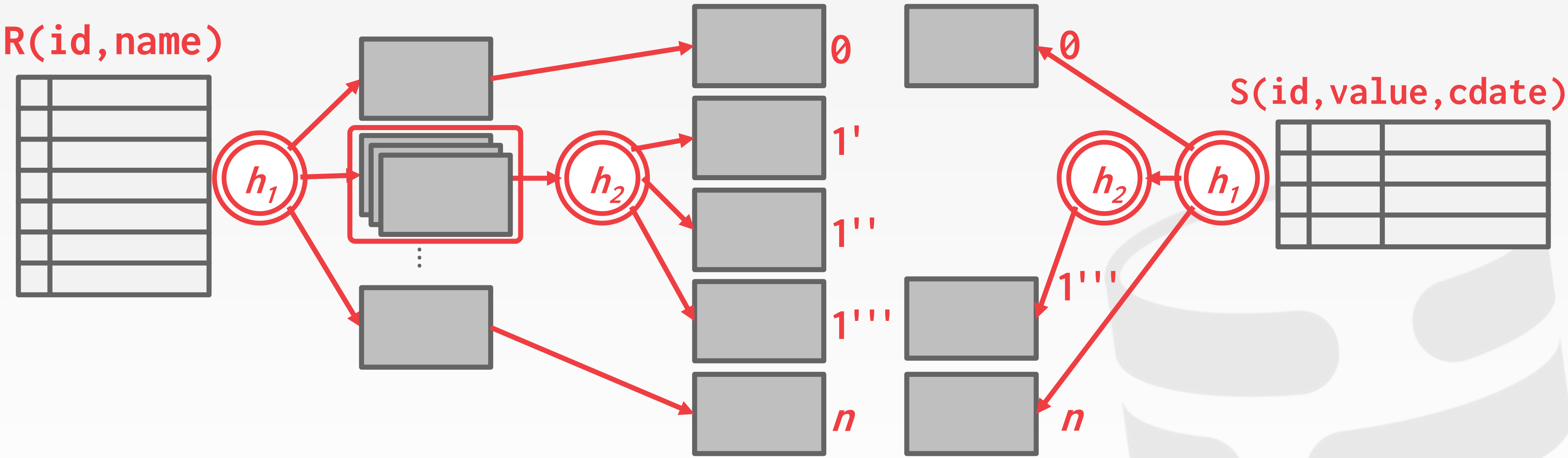
RECURSIVE PARTITIONING



RECURSIVE PARTITIONING



RECURSIVE PARTITIONING



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 ·(1000 + 500) = **4,500 IOs**

→ At 0.1 ms/IO, Total time ≈ 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 have to 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	$M + (m \cdot C)$	~20 seconds
Sort-Merge Join	$M + N + (\text{sort cost})$	0.59 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

How the DBMS decides what algorithm to use for each operator in a query plan.