Intro to Database Systems (15-445/645)

11 Join Algorithms

Carnegie Mellon University

SPRING 2023

Charlie Garrod

ADMINISTRIVIA

Homework 3 due Sunday
→ You may not turn in Homework 3 late

Midterm exam Wednesday, March 1st
→ Practice exam coming tomorrow
→ Exam accommodations? Schedule with EOS

Project 2 is available
→ First checkpoint due Friday, March 3rd (15% of P2 grade)
→ Overall due Wednesday, March 22nd (85% of P2 grade)

LAST TIME

Finished concurrent B+Trees

Sorting
→ Top-k heap sort
→ External merge sort

Aggregations
→ External hashing

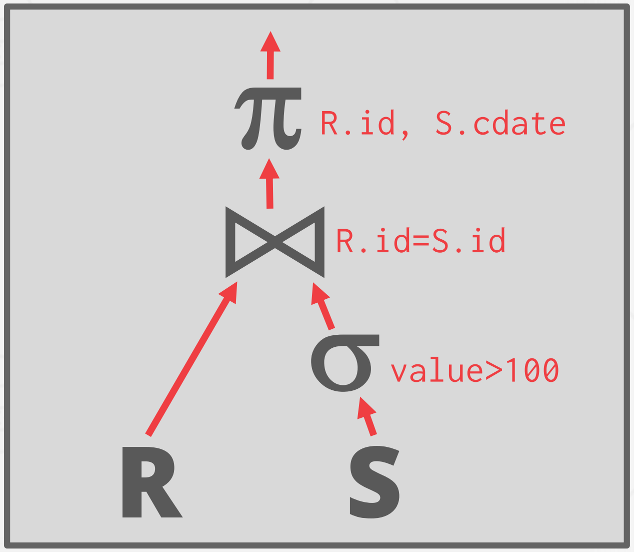
RECALL: QUERY PLAN

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
```



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.

JOIN OPERATORS

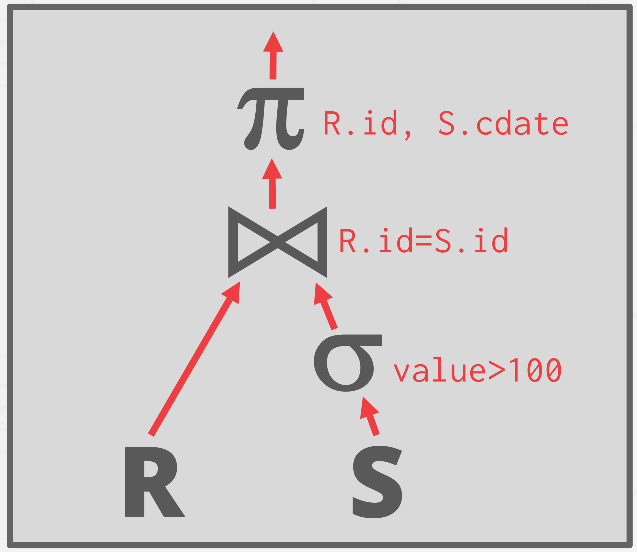
Decision #1: Output

→ 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
```

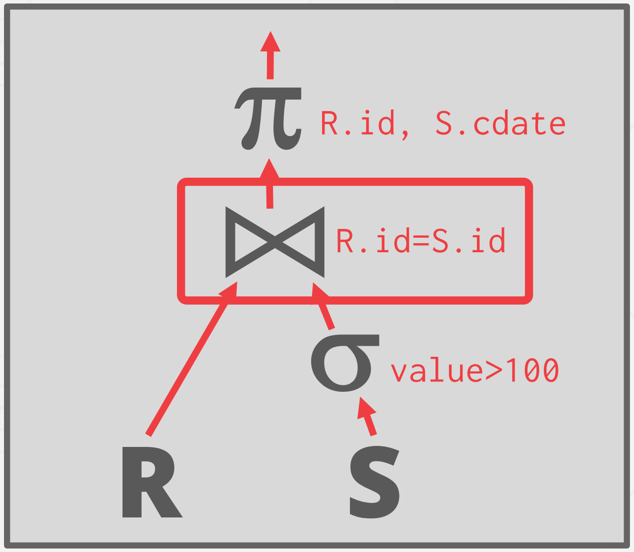


OPERATOR OUTPUT

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
```



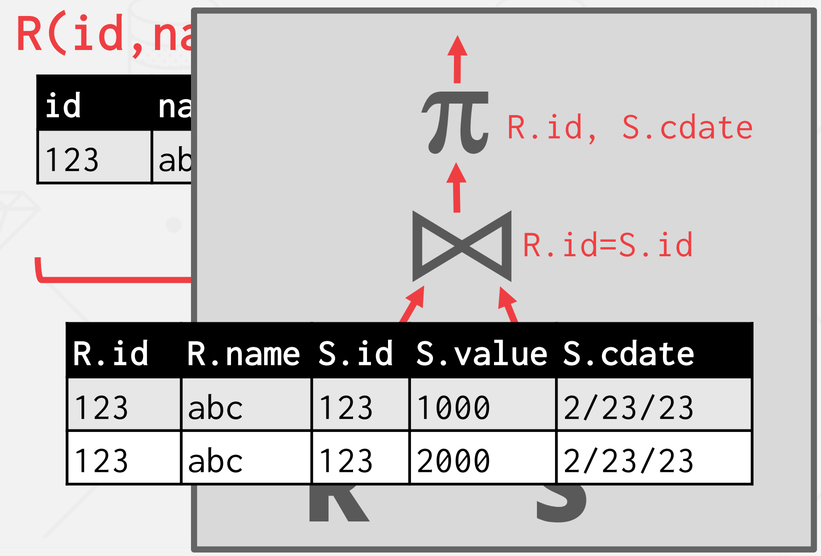
OPERATOR OUTPUT: DATA

Early Materialization:

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

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

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



R.id	R.name	S.id	S.value	S.cdate
123	abc	123	1000	2/23/23
123	abc	123	2000	2/23/23

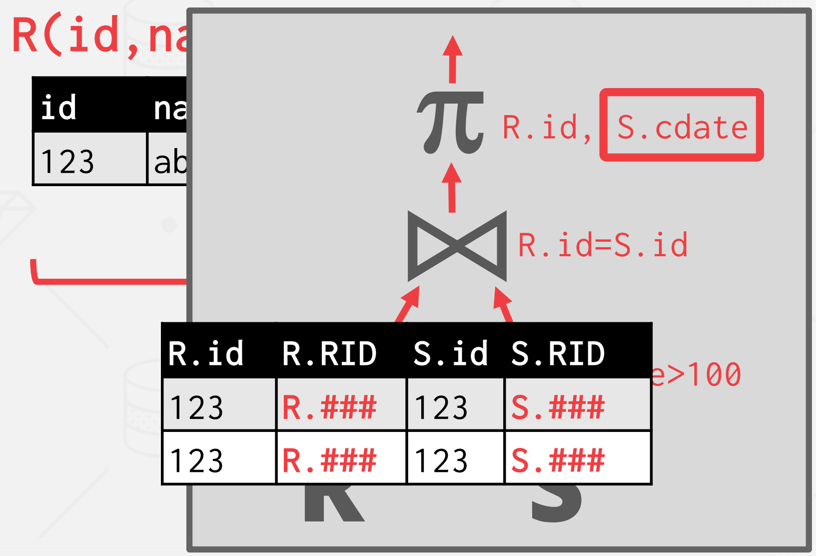
OPERATOR OUTPUT: RECORD IDS

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
```



id	na
123	ab

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

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

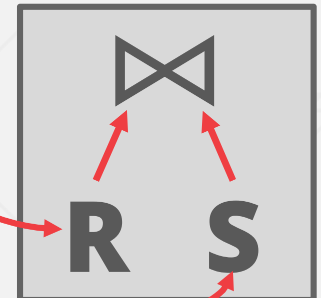
Nested Loop Join
→ Naïve
→ Block
→ Index

Sort-Merge Join

Hash Join
→ Simple
→ GRACE (Externally Partitioned)
→ Hybrid

NAÏVE NESTED LOOP JOIN

```
foreach tuple r ∈ R:      ← Outer
  foreach tuple s ∈ 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	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

NAÏVE NESTED LOOP JOIN

Why is this algorithm bad?
→ For every tuple in **R**, it scans **S** once

Cost: M + (m · 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	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

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 · 500) = **50,001,000 IOs**
→ At 0.1 ms/IO, Total time ≈ 1.3 hours

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

BLOCK NESTED LOOP JOIN

```
foreach block B_R ∈ R:
  foreach block B_S ∈ S:
    foreach tuple r ∈ B_R:
      foreach tuple s ∈ B_S:
        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	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

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 + ((# blocks in R) · 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	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

N pages
n tuples

BLOCK NESTED LOOP JOIN

The smaller table should be the outer table.

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

R(id,name)

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	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

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.

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	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

N pages
n tuples

BLOCK NESTED LOOP JOIN

If we l
→ Use
→ Use
put.

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

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	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

BLOCK NESTED LOOP JOIN

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

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

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:** $M + (\lceil M / (B\text{-}2) \rceil \cdot N)$ = 1000 + 10*500 = **6000 I/Os**
→ Or can switch inner/outer relations, giving us cost:
500 + 5*1000 = **5500 I/Os**

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.

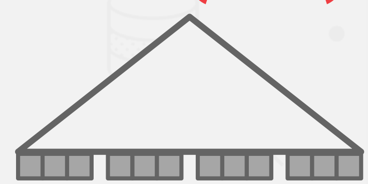
INDEX NESTED LOOP JOIN

Assu
const

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

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

Index(S.id)



R(id,name)

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

M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	2/23/23
500	7777	2/23/23
400	6666	2/23/23
100	9999	2/23/23
200	8888	2/23/23

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 (and possibly
          backtrack cursorS)
  elif cursorR and cursorS match:
    emit
    increment cursorS
```

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	2/23/23
100	9999	2/23/23
200	8888	2/23/23
400	6666	2/23/23
500	7777	2/23/23

Sort!

```
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	2/23/23
100	Andy	100	9999	2/23/23
200	GZA	200	8888	2/23/23
200	GZA	200	8888	2/23/23
400	Raekwon	200	6666	2/23/23
500	RZA	500	7777	2/23/23

SORT-MERGE JOIN

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

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

Merge Cost: $(M + N)$

Total Cost: Sort + Merge

SORT-MERGE JOIN

Example database:

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

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

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

→ Sort Cost (**R**) $= 2000 \cdot (1 + \lceil \log_{99} 1000 / 100 \rceil) =$ **4000 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 worst case for the merging phase is when the join attribute of all the tuples in both relations contains the same value.

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

WHEN IS SORT-MERGE JOIN USEFUL?

One or both tables are already sorted on join key.

Output must be sorted on join key.

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

HASH JOIN

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

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

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

SIMPLE HASH JOIN ALGORITHM

Phase #1: Build

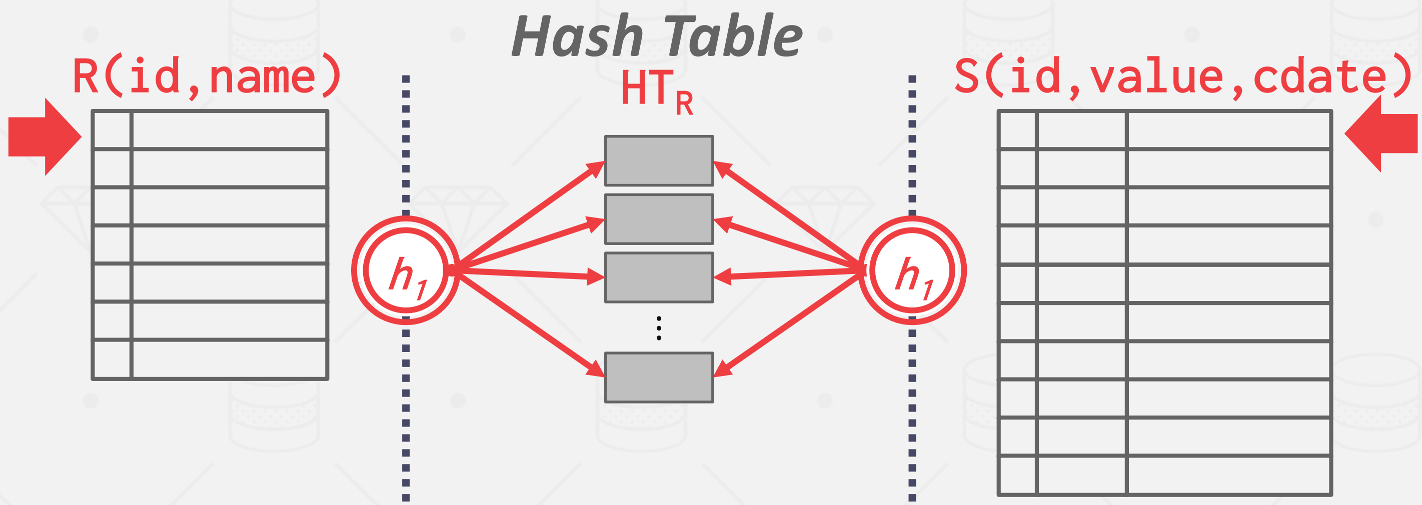
→ Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

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 ∈ S
  output, if h_1(s) ∈ 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

→ Depends on what the next query operators will do with the output from the join

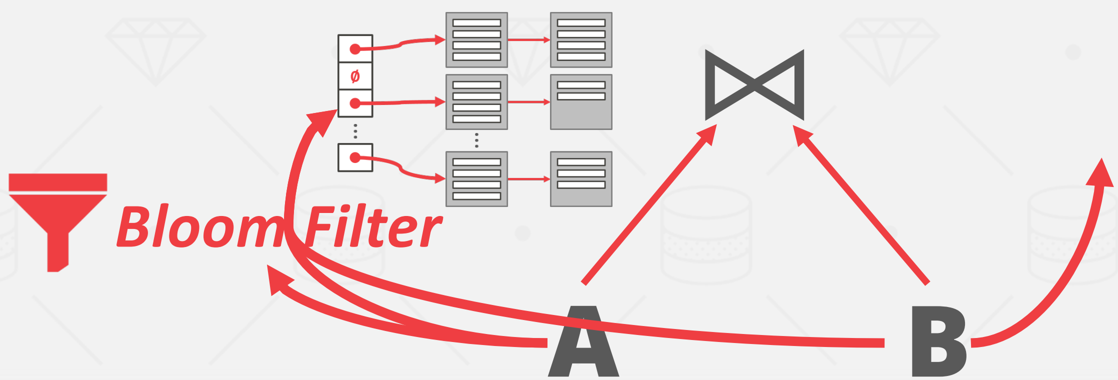
→ Early vs. Late Materialization

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



BLOOM FILTERS

Uses a bitmap to probabilistically answer set membership queries
→ False negatives will never occur
→ False positives can sometimes occur

Insert(x):
→ Use *k* hash functions to set bits in the filter to 1

Lookup(x):
→ Check whether the bits are 1 for each hash function

See the Bloom Filter Calculator if you build one

BLOOM FILTERS

Bloom Filter

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

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

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

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

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

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

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

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

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

Insert('RZA')

Insert('GZA')

Lookup(RZA') → ***TRUE***

Lookup('Raekwon') → ***FALSE***

Lookup('ODB')→ ***TRUE***

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

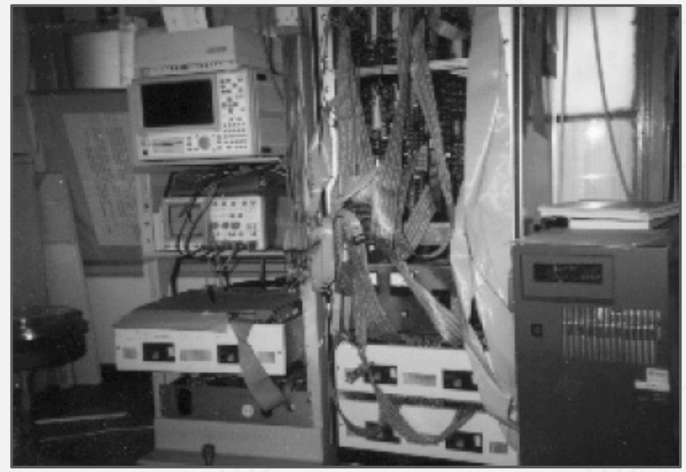
Hash join when tables do not fit in memory.

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

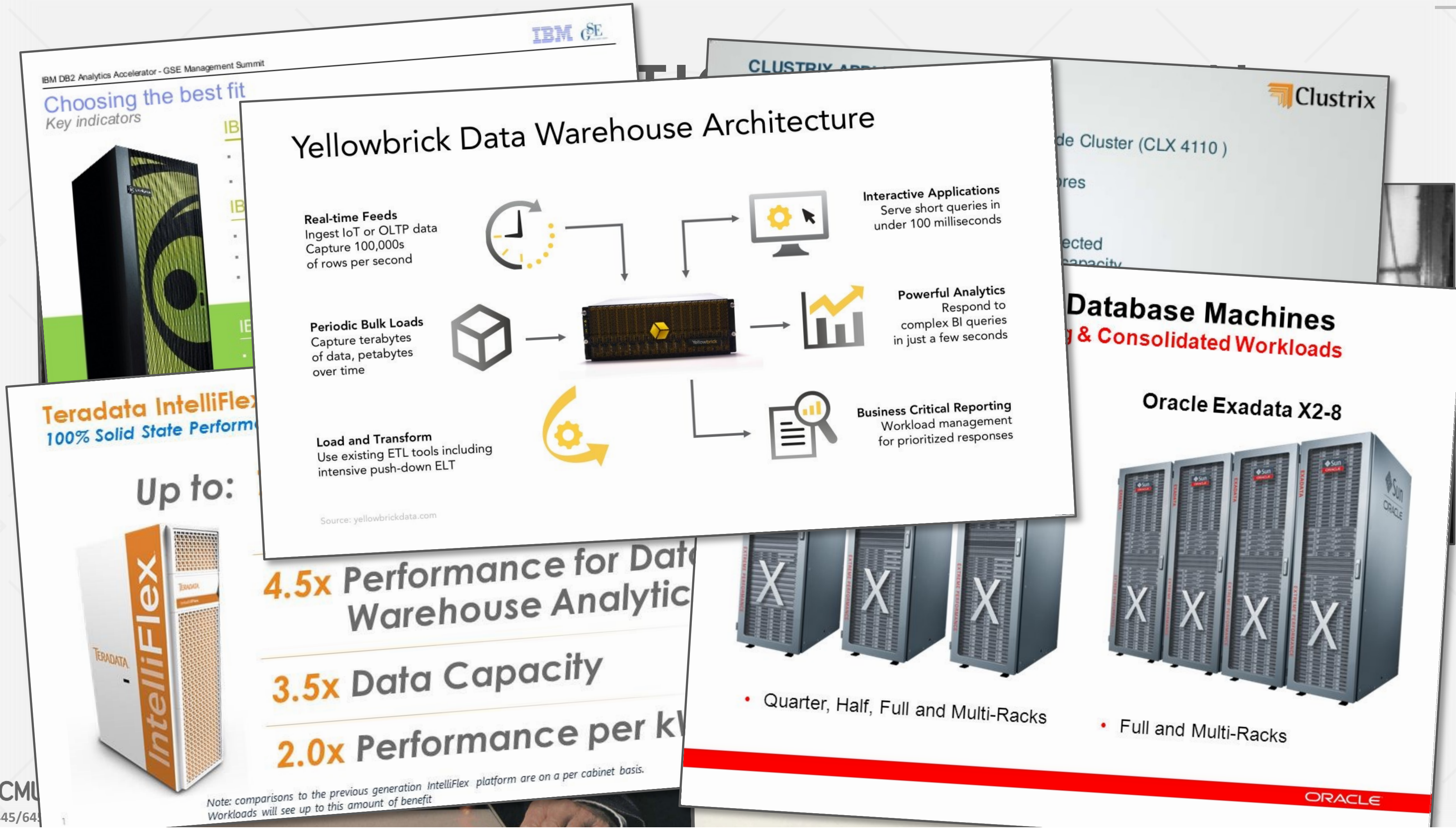
→ **Probe Phase:** Compares tuples in corresponding partitions for each table.

Sometimes called **GRACE Hash Join**.

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



GRACE
University of Tokyo



IBM DB2 Analytics Accelerator - GSE Management Summit
IBM
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
Clustrix
de Cluster (CLX 4110)
Database Machines
& Consolidated Workloads
Oracle Exadata X2-8
Quarter, Half, Full and Multi-Racks
Full and Multi-Racks
ORACLE
Teradata IntelliFlex
100% Solid State Perform
Up to:
IntelliFlex
TERADATA
4.5x Performance for Dat
Warehouse Analytic
3.5x Data Capacity
2.0x Performance per k
Note: comparisons to the previous generation IntelliFlex platform are on a per cabinet basis.
Workloads will see up to this amount of benefit

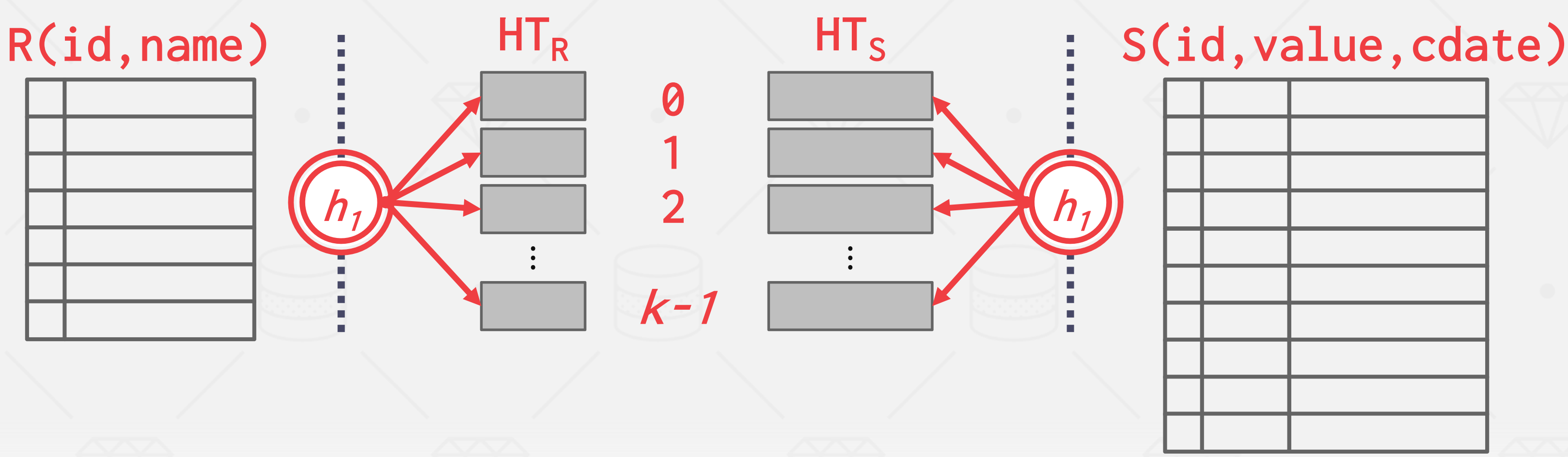
PARTITIONED HASH JOIN PARTITION PHASE

因为需要读入input和写回，所以
这个时候size最大B-1
如果不需要则 B

Hash **R** into *k* buckets.

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

Write buckets to disk when they get full.

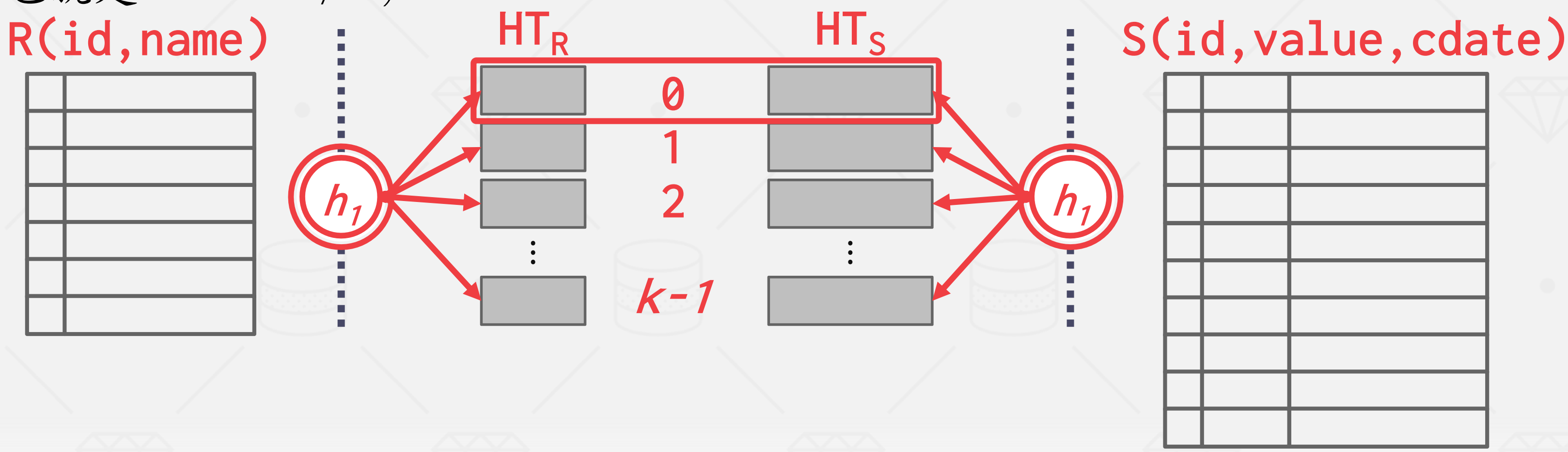


PARTITIONED HASH JOIN PROBE PHASE

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

针对每一个partition，做同等的hash，也就是把从p(rs)hash成b-2个块(p(rs)的“size”要< b-2!)，另一个读，另一个写出结果，如果不需要则b-1

（也就是 M-1 > bs/M）



PARTITIONED HASH JOIN EDGE CASES

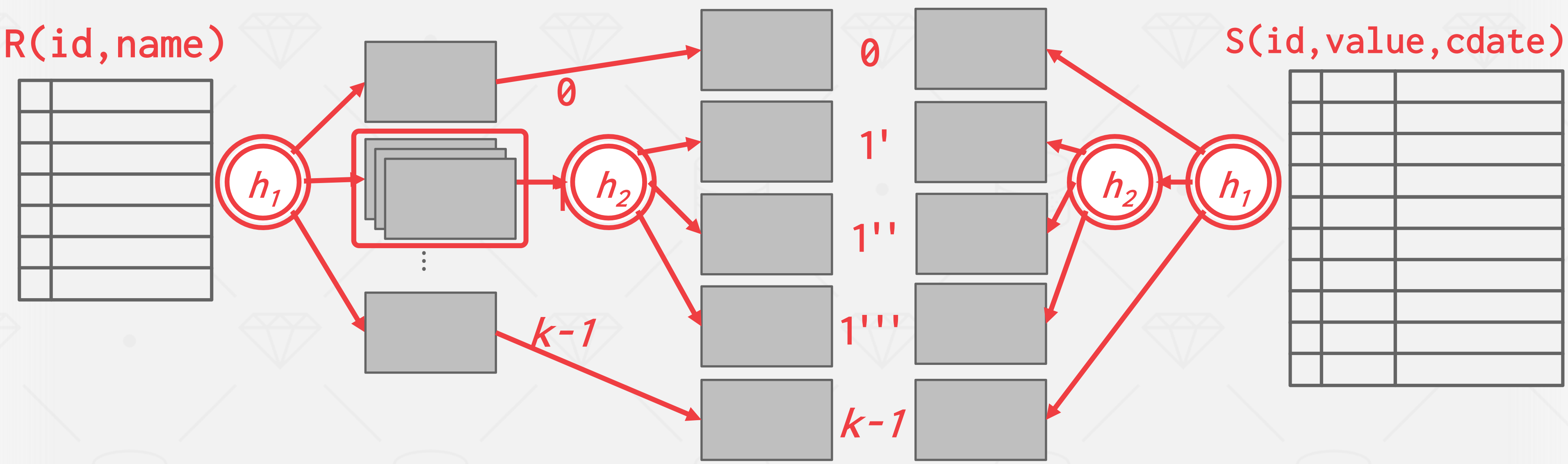
If a partition does not fit in memory, recursively partition it with a different hash function

→ 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



ANALYSIS OF PARTITIONED HASH JOIN

How big a table can be joined without recursive partitioning?
→ Up to ***B*-1** partitions
→ Each could be about as big as ***B-2*** pages

Answer: About **(B-1) · (B-2)** pages
→ If the partitions are approximately equal size, a table of ***N*** pages needs about **sqrt(*N*)** buffers
→ In practice, use a "fudge factor" ***f* > 1**: **sqrt(*f* · *N*)**
→ Only partitions of the outer table need to fit in memory

COST OF PARTITIONED HASH JOIN

If we don't 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

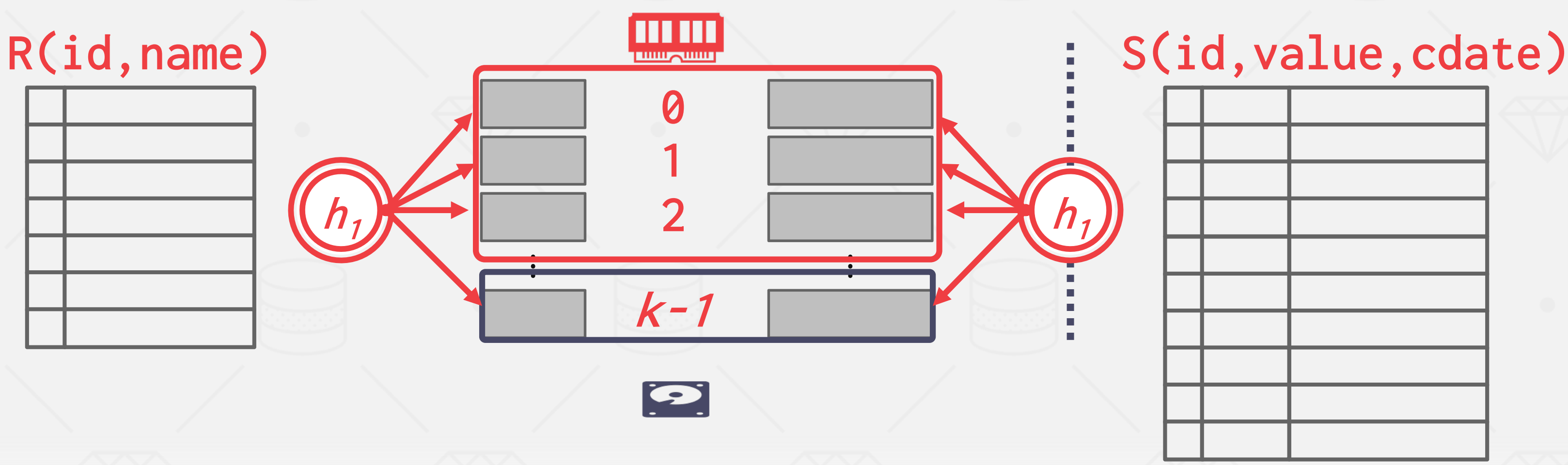
PARTITIONED HASH JOIN

Example database:
→ $M = 1000$, $m = 100{,}000$
→ $N = 500$, $n = 40{,}000$

Cost Analysis:
→ $\mathbf{3 \cdot (M + N)} = 3 \cdot (1000 + 500) = \mathbf{4{,}500\ IOs}$
→ At 0.1 ms/IO, Total time ≈ 0.45 seconds

OPTIMIZATION: HYBRID HASH JOIN

Use some buckets for a simple in-memory hash join, have some buckets spill to disk.



HASH JOIN OBSERVATIONS

The inner table can be any size.
→ Only outer table (or its partitions) need to fit in memory

If we know the size of the outer 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	$M + (\lceil M / (B\text{-}2) \rceil \cdot N)$	0.55 seconds
Index Nested Loop Join	$M + (m \cdot C)$	Variable
Sort-Merge Join	$M + N$ + (sort cost)	0.75 seconds
Hash Join	$3 \cdot (M + N)$	0.45 seconds

CONCLUSION

Hashing is almost always better than sorting for operator execution.

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

Good DBMSs use either (or both).

NEXT CLASS

Composing operators together to execute queries.