COMP 421: Files & Databases

Lecture 12: Joins

UNC
DEPARTMENT OF
COMPUTER SCIENCE

Announcements

Mid-term Exam on Monday Oct 20th

→ In-class in this room.
→ Review session in class on Oct. 15th
→ Format: 5 Big Questions: Q1 will have some multiple choice, other questions mostly open response
→ Know the material, but do not focus on memorizing every minute detail, we will try and test ***concepts.***
→ Know design, guarantees, runtimes of algos and data structures from lecture
→ Textbook provides many good problem with solutions for practice

Come to class Wednesday with specific questions! I will pull up slides and explain anything you want in greater detail. No questions? We'll talk about more data structures.

Last Week

Last week was data structure week!

- In addition to previously discussed B+Tree...
- Bloom Filters
- Skip Lists

How to make these things concurrent/thread-safe

- Atomic instructions (compare-and-swap)
- OS-level mutexes (futex)
- Reader-writer latches
- Latch protocols (e.g., latch crabbing for B+Trees)

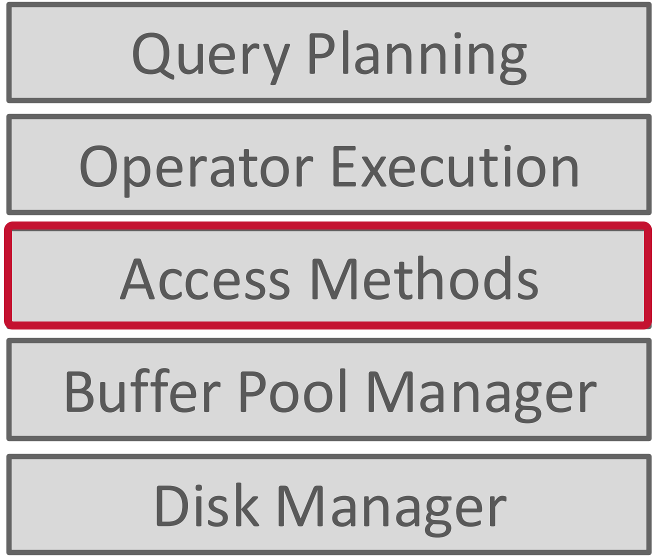
Course Progress Check-In

We are done with Access Methods!

Operator Execution

- How to translate relational operators into fast code
- Good news: one of the hard ones, sorting, we've already done
- **Today**: the other hard one, **joins**

Goal: want fast, I/O efficient operators to use when we get to query planning and execution



Why Do We Need To Join?

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

We then use the **<u>join operator</u>** 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 (e.g., worst-case optimal joins).

In general, join algorithms work better when we can identify the smaller of the two tables

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

Query Plan

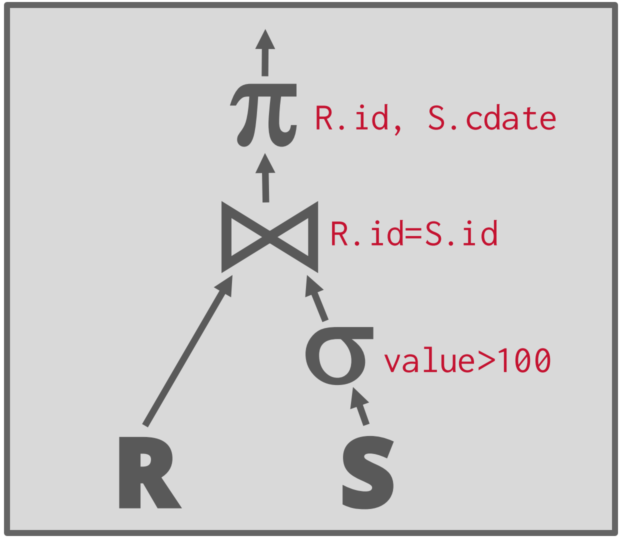
The operators are arranged in a tree.

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

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

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

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



Join Operators

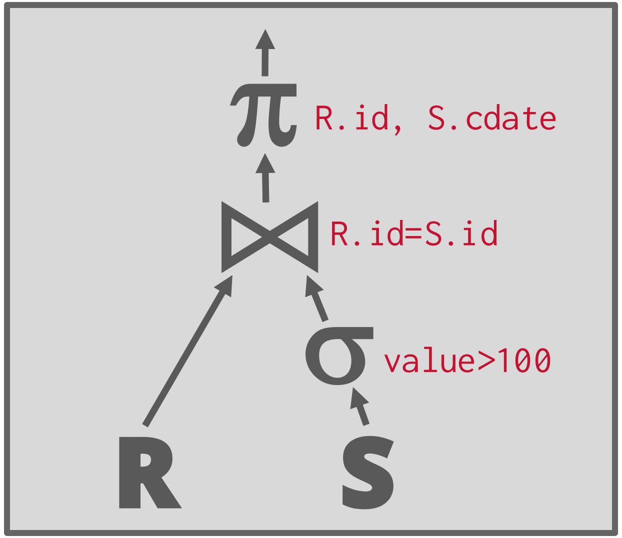
Decision #1: Output

→ For each row of the join, what data is emitted to the parent operator in the query plan tree?

Decision #2: Cost Analysis Criteria

→ How to design/choose an algorithm to identify which rows to emit?

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

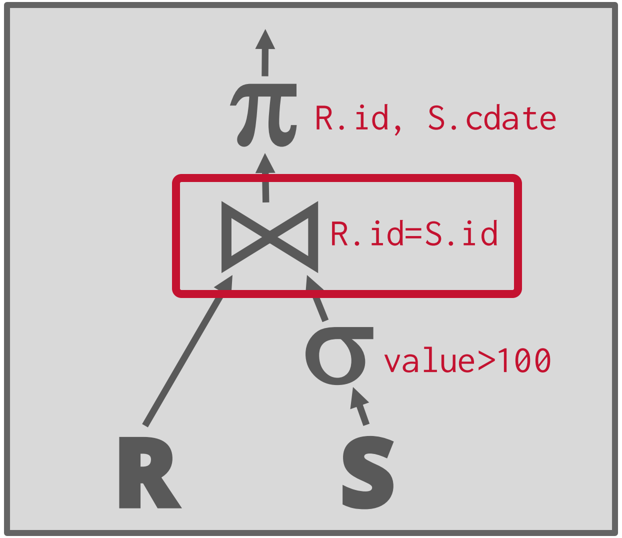


Operator Output

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
```



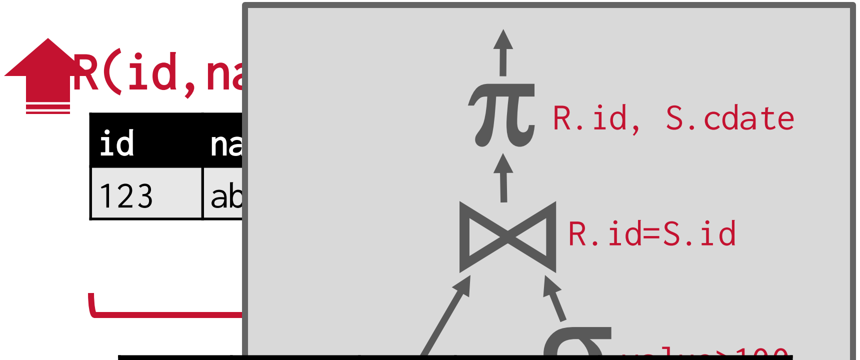
Operator Output: Tuple Data

Early Materialization:

→ Copy values for the attributes in outer and inner tuples into 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	10/13/2025
123	abc	123	2000	10/13/2025

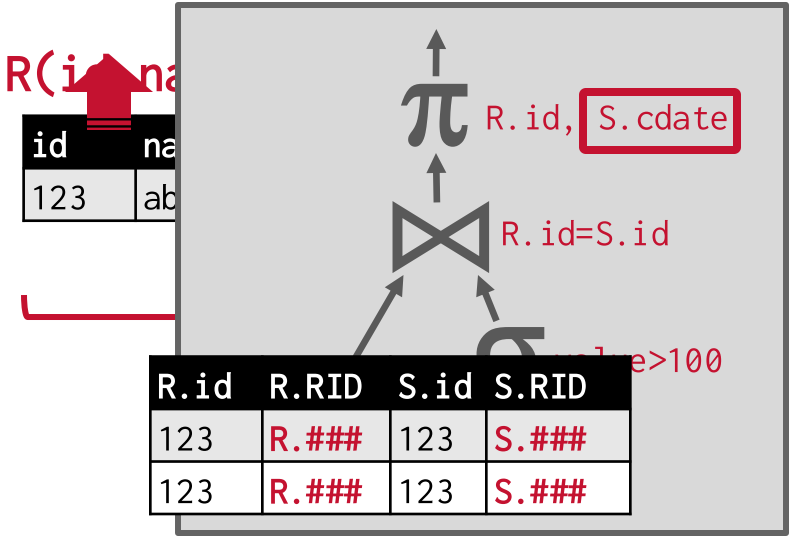
Operator Output: Record IDs

Late Materialization:

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

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

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



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

What Algorithm to Use?

Given a query that joins table **R** with table **S**, assume the DBMS has the following information those tables:

→ ***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

→ Ignore result output costs because it depends on the data and is the same for all algorithms.

→ Ignore computation / network costs (for now).

→ When sequential vs. random I/O is an issue, I'll point it out

Join Algorithms

Nested Loop Join

→ Naïve
→ Block
→ Index

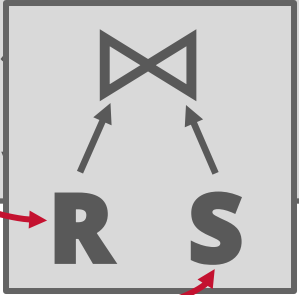
Sort-Merge Join

Hash Join

→ Simple
→ GRACE (Externally Partitioned)
→ Hybrid

Naïve Nested Loop Join

```
SELECT R.id, S.cdate
  FROM R
  R.
  S.
```



Don't Do This!

```
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	10/13/2025
500	7777	10/13/2025
400	6666	10/13/2025
100	9999	10/13/2025

Naïve Nested Loop Join

Why is this algorithm bad?

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	10/10/25
500	7777	10/10/25
400	6666	10/10/25
100	9999	10/10/25
200	8888	10/10/25

Naïve Nested Loop Join

Why is this algorithm bad?

→ Read every tuple in **R**

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

Cost: M + (m · 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	10/13/2025
500	7777	10/13/2025
400	6666	10/13/2025
100	9999	10/13/2025

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 · 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
```

All in memory!

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	10/10/25
500	7777	10/10/25
400	6666	10/10/25
100	9999	10/10/25
200	8888	10/10/25

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)

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	10/10/25
500	7777	10/10/25
400	6666	10/10/25
100	9999	10/10/25
200	8888	10/10/25

N pages
n tuples

Block Nested Loop Join

The smaller table should be the outer table.

Compare: $M + (M \cdot N)$ vs. $N + (N \cdot M)$

It turns out this also improves sequential I/O

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	10/10/25
500	7777	10/10/25
400	6666	10/10/25
100	9999	10/10/25
200	8888	10/10/25

N pages
n tuples

UNC DEPARTMENT OF COMPUTER SCIENCE

What About The Buffer Pool?

If w

→ U e.

→ U output.

```
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
```

All in memory!

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	10/10/25
500	7777	10/10/25
400	6666	10/10/25
100	9999	10/10/25
200	8888	10/10/25

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 ($B-2>M$):

→ **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**

→ Total time ≈ 0.55 seconds

Nested Loop Join

Why is the basic nested loop join so bad?

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

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

id	value	cdate
100	2222	10/10/25
500	7777	10/10/25
400	6666	10/10/25
100	9999	10/10/25
200	8888	10/10/25

Quadratic # of comparisons to find a linear number of matches

→ Lots of wasted work

→ **Idea:** Data structure / algorithm to find matches with fewer comparisons

Index Nested Loop Join

Assume the cost of each index lookup is ***C*** per tuple.

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

M pages
m tuples

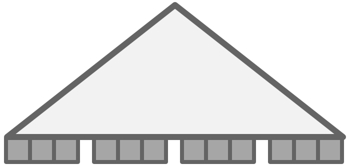
R(id,name)

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	10/13/2025
500	7777	10/13/2025
400	6666	10/13/2025
100	9999	10/13/2025

Index(S.id)



N pages
n tuples

Index Nested Loop Join

The hidden cost of index nested loop joins: **random I/O**

Block nested loop join: # Disk Seeks = $2M$

Index nested loop join: # Disk Seeks = $M + m \cdot C_{seek}$

R(id,name)

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

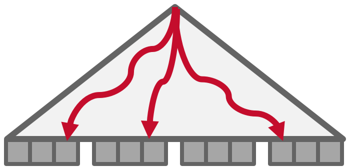
M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	10/10/25
500	7777	10/10/25
400	6666	10/10/25
100	9999	10/10/25
200	8888	10/10/25

N pages
n tuples

Index(S.id)



Index Nested Loop Join

Assume 0.1 ms I/Os and 4 ms se

Block nested loop join (minimal

→ # I/Os = $N + (N \cdot M)$ = 500,500 I/O

→ # Disk Seeks = $2N$ = 1000 seeks

→ Total time ≈ 50 seconds + 4 seco

Index nested loop join using a B+Tree

→ # I/Os = $N + (n \cdot 5)$ = 1000 + 200,000 = 200,100 I/Os

→ # Disk Seeks = $N + (n \cdot 5)$ = 200,100 seeks

→ Total time ≈ ≈ 20 seconds + **800 seconds = 820 seconds**

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

What to improve?

Index-based method didn't "glue together" the index and the algorithm. Need to **jointly optimize** algorithm and data structure.

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 to handle duplicates

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	10/13/2025
100	9999	10/13/2025
200	8888	10/13/2025
400	6666	10/13/2025
500	7777	10/13/2025

Last Value: 400

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	10/13/202
100	Andy	100	9999	10/13/202
200	GZA	200	8888	10/13/202
200	GZA	200	8888	10/13/202
400	Raekwon	200	6666	10/13/202
500	RZA	500	7777	10/13/2025

Sort-Merge Join Cost (Best Case)

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 Cost (Best Case)

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 Cost (Worst Case)

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.

Observation

Sort-Merge Join pros and cons:

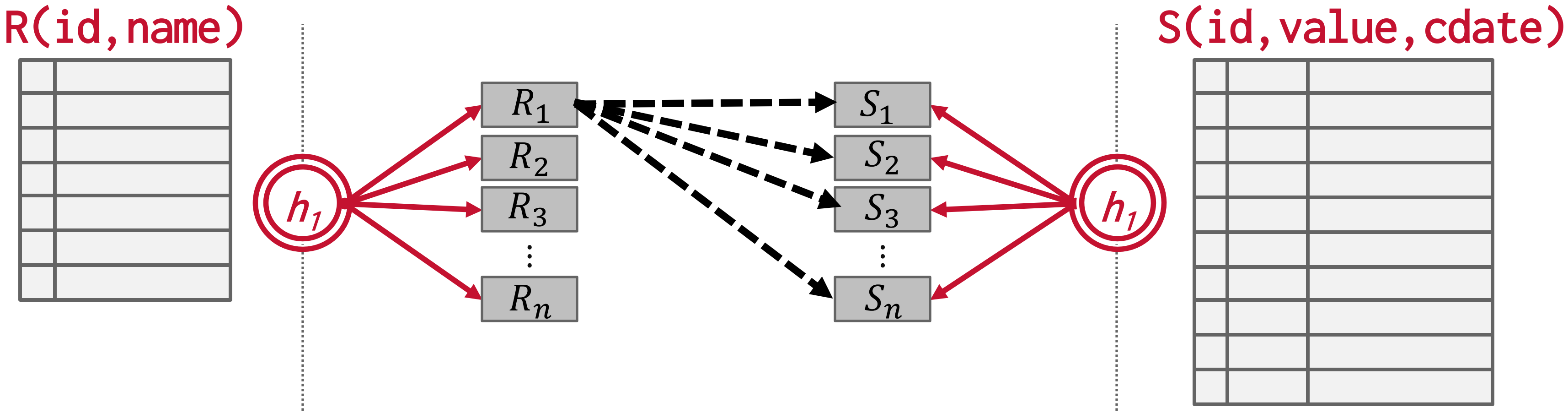
- ✓ Used block-based layout and
- ✓ Could take advantage of buffer pool
- X Still suffered from indexing overhead
- X Still suffered from excessive random I/O

Hash join tries to avoid these issues

- ✓ Make use of blocks/buffers
- ✓ Mostly sequential I/O
- ✓ Low indexing overhead

Hash Join

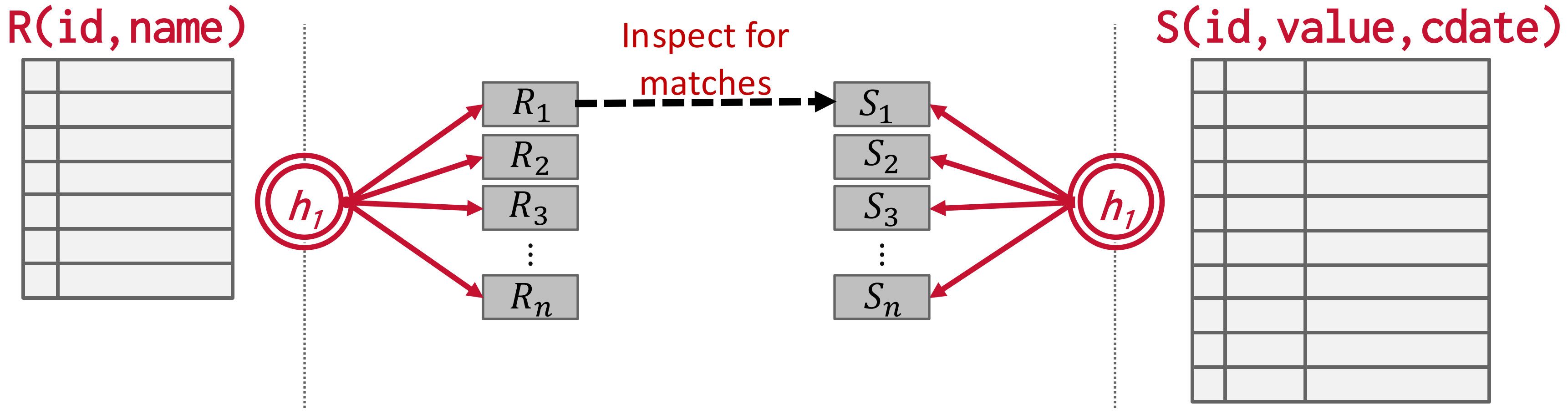
Idea: Hash tuples into n buckets based on join key (e.g., id)



For $r \in R_i$ and $s \in S_j$, $h_1(r.id) \neq h_1(s.id)$, so we don't need to compare R_i to S_j

Hash Join

Idea: Hash tuples into n buckets based on join key (e.g., id)



For $r \in R_i$ and $s \in S_i$, $h_1(r.id) = h_1(s.id)$, so either we had a hash collision, or $r.id = s.id$

Simple Hash Join Algorithm

Phase #1: Build

→ Scan the outer relation and populate a hash table, HT, 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.

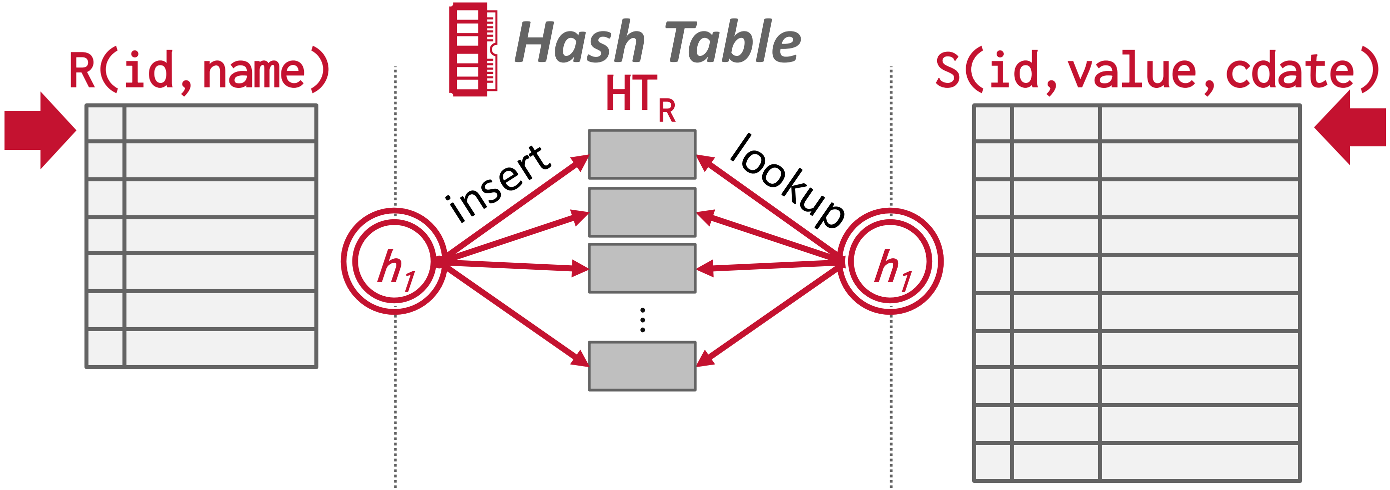
Phase #2: Probe

→ 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.

For starters, assume HT fits in memory on B buffers

Simple Hash Join Algorithm

```
foreach tuple r ∈ R:
  insert h1(r) into hash table HT_R
foreach tuple s ∈ S:
  output, if h1(s) ∈ HT_R
```



Hash Joins Of Large Relations

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

Buffer pool manager might swap out pages of HT at random

Need a principled approach to minimize I/O in this case

Partitioned Hash Join

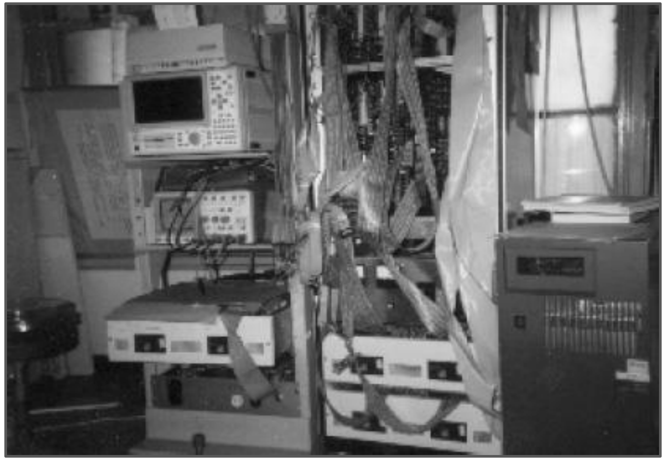
Hash join when tables do not fit in memory.

→ **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.



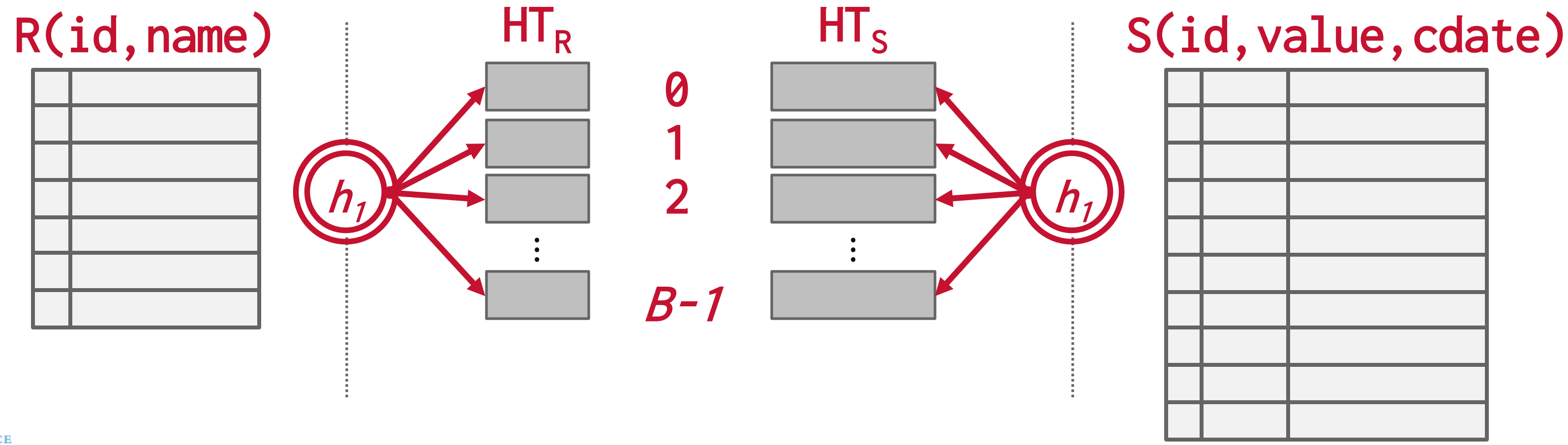
GRACE
University of Tokyo

Partitioned Hash Join (Partition Phase)

Hash R into *B* buckets.

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

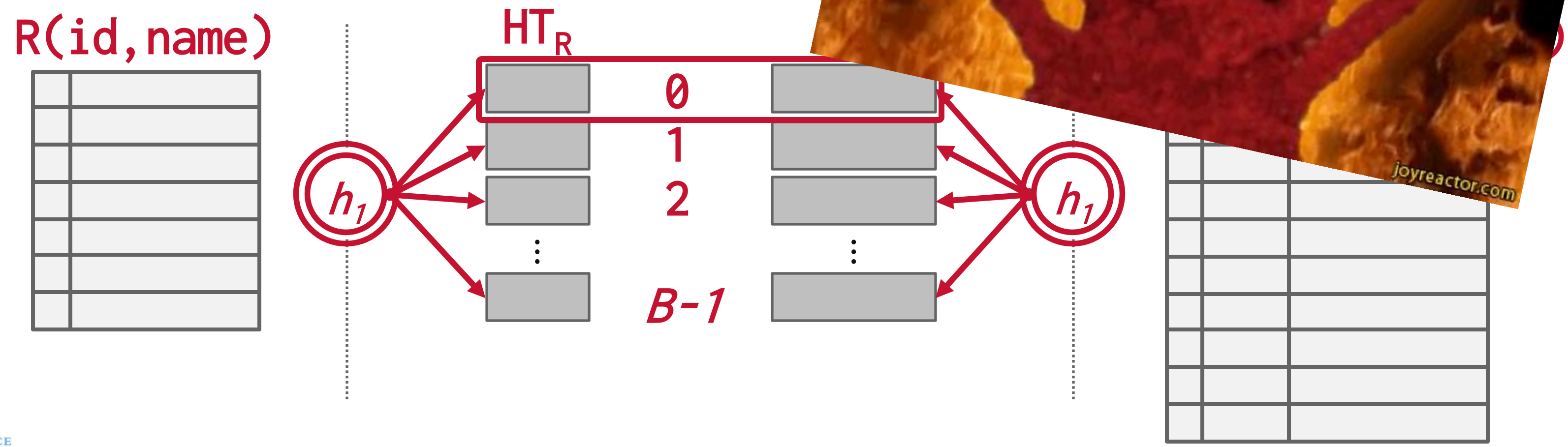
Write buckets to disk when they get full.



Partitioned Hash Join Probe Phase

Read corresponding partitions in… one pair at a time, use simple hash join on th…

But wait, R_i might have many …

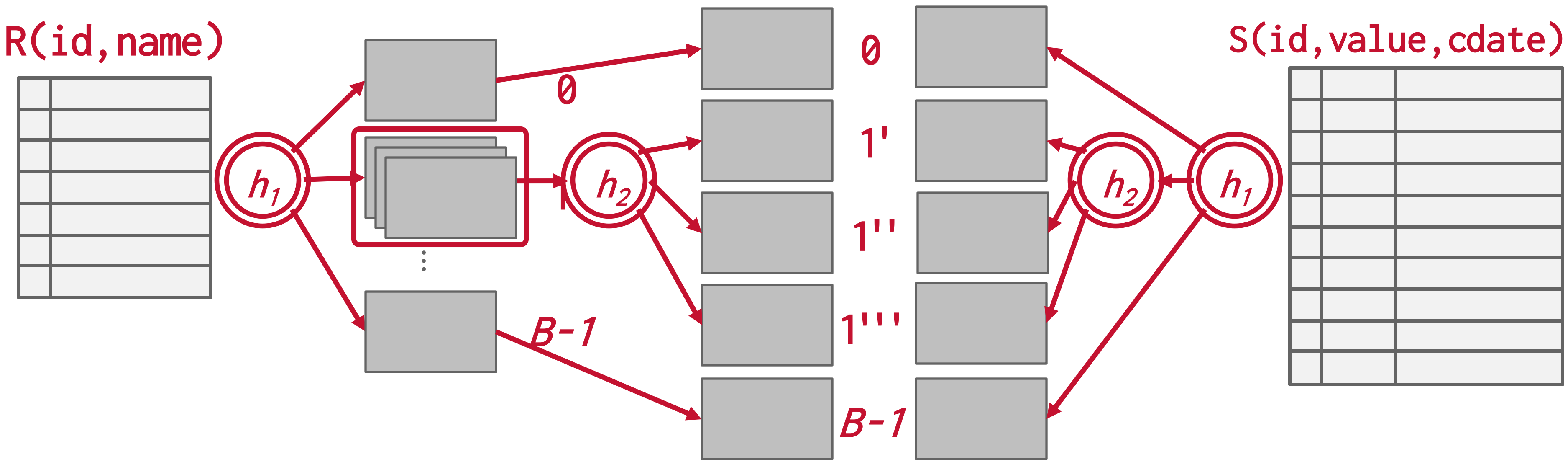


Partitioned Hash Join Edge Cases

Option 1: If a single join key has too many matching records that do not fit in memory, use a **block nested loop** join just for that key.
→ Avoids random I/O in exchange for sequential I/O.

Option 2: 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

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:

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

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

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	$M + (m \cdot C)$	>20 seconds
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 many/all approaches when needed

Next Class

Midterm Review

Come with questions!

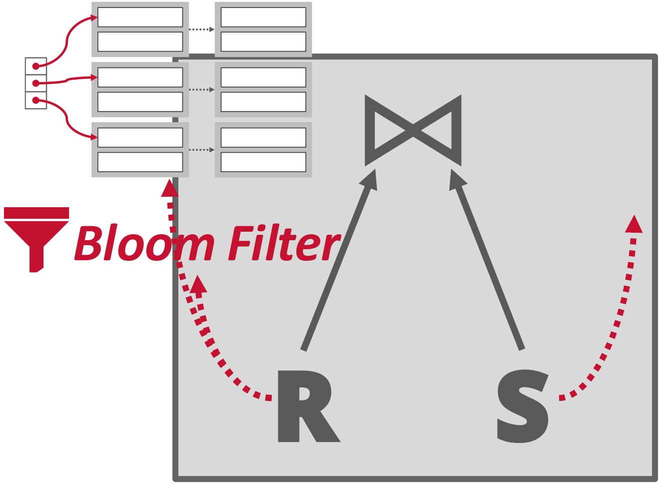
Optimization: Probe Filter

Create a probe filter (Bloom Filter) as the DBMS builds the hash table on the outer table in the first phase.

→ Always check the filter before probing the hash table.

→ Faster than probing hash table because the filter fits in CPU cache.

This technique is sometimes called ***sideways information passing.***



Optimization: Hybrid Hash Join

If the keys are skewed, then the DBMS keeps the hot partition in-memory and immediately perform the comparison instead of spilling it to disk.

→ Difficult to get to work correctly. Rarely done in practice.