Advanced Databases

Spring 2020

Lecture #09:

Join Algorithms

Milos Nikolic

THE UNIVERSITY of EDINBURGH

Why Do We Need to Join?

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

We use the join operator to reconstruct the original tuples without any information loss

Normalised Tables

Course(cid, name, dept)

cid	name	dept
11011	Advanced Databases	CS
11122	Database Theory	CS
10070	Database Systems	CS

Enrolled(cid, sid)

cid	sid
11011	123466
11122	123488
10070	123488
11122	123466

Student(sid, name, gender)

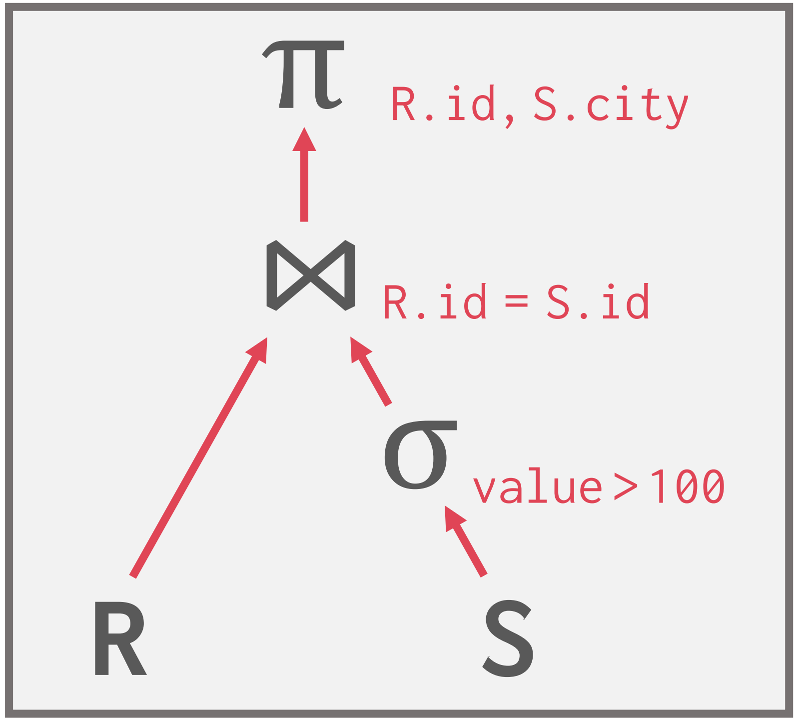
name	name	gender
123466	Alice	F
123488	Michael	M

Join Operator Output

For a tuple $r \in R$ and a tuple $s \in S$ that match on join attributes, concatenate r and s together into a new tuple

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

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



Joins: Overview

Joins are among the most **expensive** operations

- # of joins often used as a measure of query complexity
- Join of 10s of tables common in enterprise apps

Naïve implementation: $R \bowtie_c S \equiv \sigma_c(R \times S)$

- Enumerate the cross product, then filter using the join condition
- Inefficient because the cross product is large

Three classes of join algorithms:

- Nested loops
- Sort-merge
- Hash

No particular algorithm works well in all scenarios

I/O Cost Analysis

Assume:

- Table **R** has ***M*** pages and ***m*** tuples in total
- Table **S** has ***N*** pages and ***n*** tuples in total

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

Cost Metric: # of I/Os to compute join

- Ignore output costs (same for all join algorithms)
- Ignore CPU costs

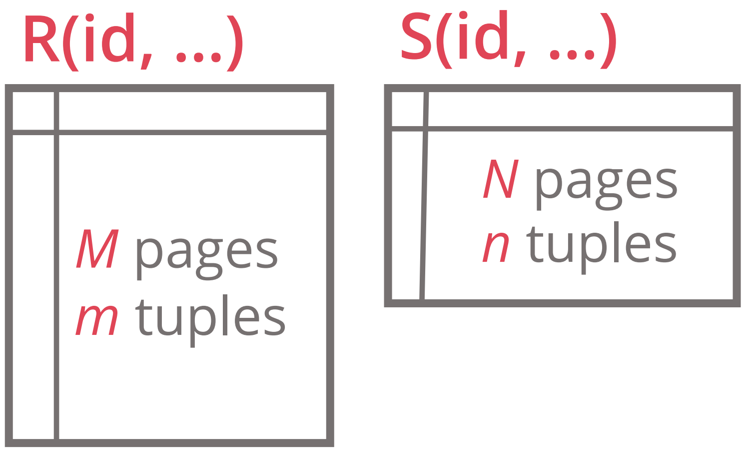
Simple Nested Loops Join


```
foreach tuple r ∈ R:        ← Outer table
  foreach tuple s ∈ S:      ← Inner table
    emit if r and s match
```

Why is this algorithm bad?

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

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



Simple Nested Loops Join

Example database:

$M = 1000$, $m = 100{,}000$

$N = 500$, $n = 40{,}000$

Cost analysis:

$M + (m \cdot N) = 1000 + (100{,}000 \cdot 500) =$ **50,001,000 I/Os**

At 0.1ms per I/O, total time ≈ **1.4 hours**

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

$N + (n \cdot M) = 500 + (40{,}000 \cdot 1000) =$ **40,000,500 I/Os**

At 0.1ms per I/O, total time ≈ **1.1 hours**

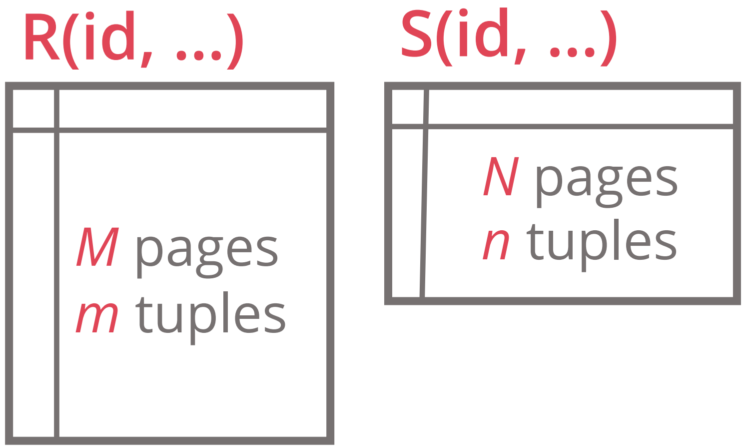
Block Nested Loops Join

```
foreach block b_R ∈ R:
  foreach block b_S ∈ S:
    foreach tuple r ∈ b_R:
      foreach tuple s ∈ b_S:
        emit if r and s match
```

This algorithm makes fewer disk accesses

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

Cost: $M + (M \cdot N)$ (*block* = *page*)



Block Nested Loops Join

Example database:

M = 1000, m = 100,000

N = 500, n = 40,000

Which one should be the outer table?

The smaller table in terms of # of pages

Cost analysis:

$N + (M \cdot N)$ = 500 + (1000 · 500) = **500,500 I/Os**

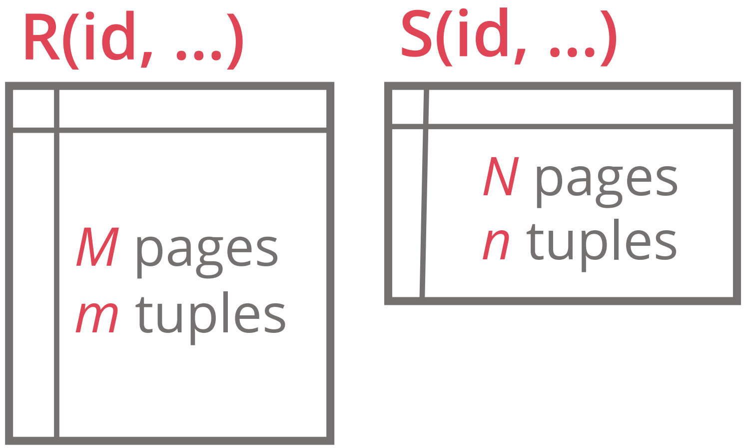
At 0.1ms per I/O, total time ≈ **50 seconds**

Block Nested Loops Join

```
foreach B-2 block b_R ∈ R:
  foreach block b_S ∈ S:
    foreach tuple r ∈ b_R:
      foreach tuple s ∈ b_S:
        emit if r and s match
```

What if we have ***B*** buffers available?

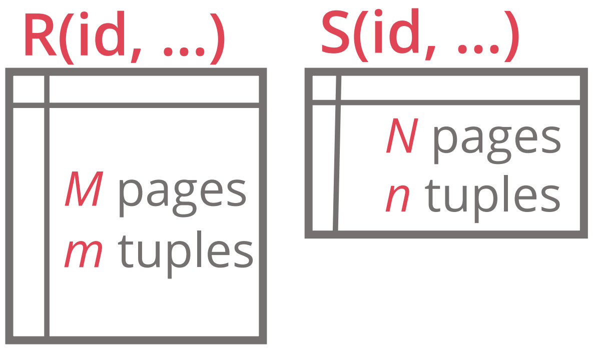
- ***B-2*** buffers for scanning the outer table
- *1* buffer for scanning the inner table
- *1* buffer for storing the output



Block Nested Loops Join

```
foreach B-2 block b_R ∈ R:
  foreach block b_S ∈ S:
    foreach tuple r ∈ b_R:
      foreach tuple s ∈ b_S:
        emit if r and s match
```

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



If the outer relation (R) fits in memory ($B \geq M + 2$)

Cost: $M + N$ = 1000 + 500 = 1500 I/Os   (*optimal cost*)

At 0.1ms per I/O, total time ≈ **0.15 seconds**

Index Nested Loops Join

Why do simple nested loops joins suck?

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

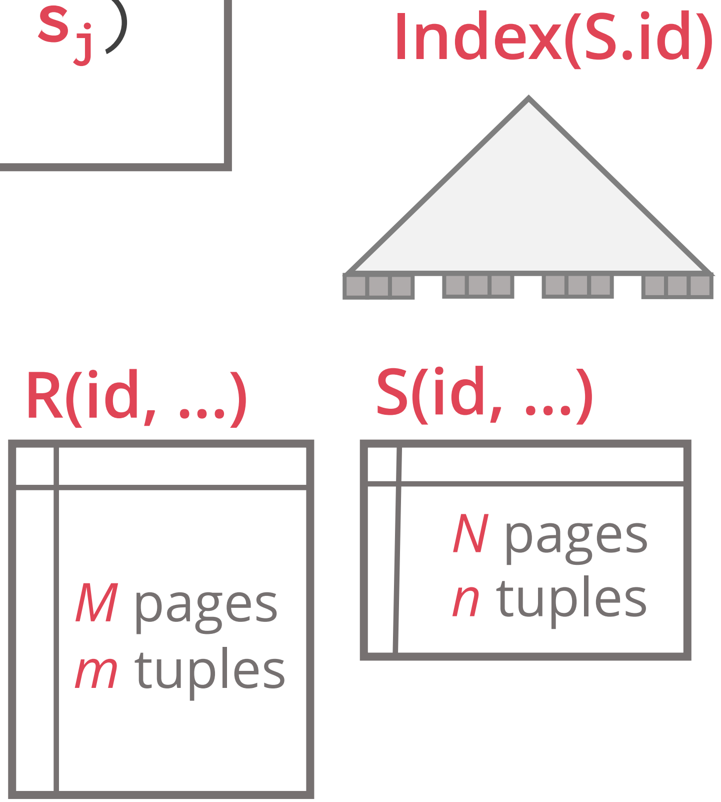
Can we accelerate the join using an index?

Use an index to find inner tuple matches

- We could use an existing index or even build one on the fly
- The index must match the join condition

Index Nested Loops Join

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



Cost: $M + (m \cdot$ cost of finding all matches in S$)$

Index access cost per R tuple:

- B+ tree: 2-4 I/Os to reach a leaf + fetch matching S tuples
- Hash index: 1.2 I/Os to reach the target bucket

The cost depends on the size of the join result

- Using an index pays off if the join is **selective**

Nested Loops Joins

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)

E.g. by using the external merge sort algorithm or by scanning the table using an index on the join key

Phase #2: Merge

Scan the two sorted tables in parallel and emit matching tuples

Sort-Merge Join

```
sort R,S on join key A
r ← position of first tuple in R_sorted
s ← position of first tuple in S_sorted
while r ≠ EOF and s ≠ EOF:
  if r.A > s.A:
    advance s
  else if r.A < s.A:
    advance r
  else if r.A = s.A:
    emit (r,s)
    advance s
```

assumes no duplicates in R
(the merge phase could be easily
extended to support duplicates)

Sort-Merge Join

R(id, name)

id	name
600	Daniel
200	Michael
100	Alice
300	Bob
500	Carrol
700	Lucia
400	John

Sort!

S(id, value, city)

id	value	city
100	2222	Edinburgh
500	7777	Edinburgh
400	6666	London
100	9999	London
200	8888	Oxford

Sort!

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

SORT-MERGE JOIN

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

Sort!

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

Sort!

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

SORT-MERGE JOIN

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Sort-Merge Join

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh

Sort-Merge Join

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London

Sort-Merge Join

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London

Sort-Merge Join

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London
200	Michael	200	8888	Oxford

SORT-MERGE JOIN

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London
200	Michael	200	8888	Oxford

SORT-MERGE JOIN

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London
200	Michael	200	8888	Oxford

Sort-Merge Join

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London
200	Michael	200	8888	Oxford
400	John	400	6666	London

Sort-Merge Join

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
$\boxed{400}$	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
$\boxed{500}$	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London
200	Michael	200	8888	Oxford
400	John	400	6666	London

Sort-Merge Join

R(id, name)

id	name
100	Alice
200	Michael
300	Bob
400	John
500	Carrol
600	Daniel
700	Lucia

S(id, value, city)

id	value	city
100	2222	Edinburgh
100	9999	London
200	8888	Oxford
400	6666	London
500	7777	Edinburgh

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

Output Buffer

R.id	R.name	S.id	S.value	S.city
100	Alice	100	2222	Edinburgh
100	Alice	100	9999	London
200	Michael	200	8888	Oxford
400	John	400	6666	London
500	Carrol	500	7777	Edinburgh

Sort-Merge Join

Lecture #07

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

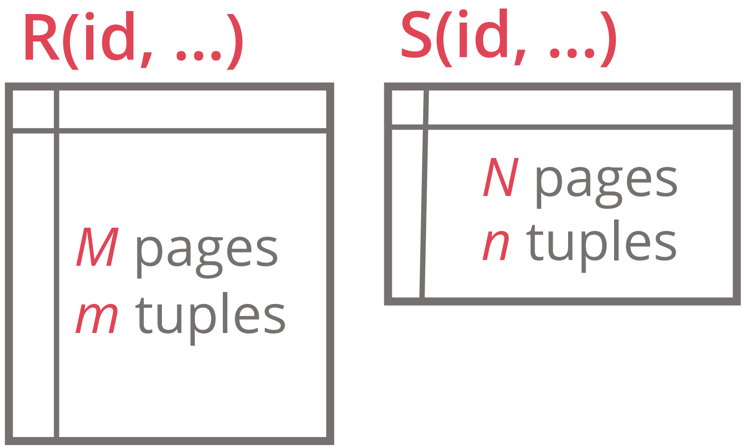
Sort Cost (S) = $2N \cdot (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$ (= $2N \cdot$ # of passes)

Merge Cost: $M + N$

The worst case for the merging phase is when the join attribute of all the tuples in both relations contain the same value

Merge Cost: $M + M \cdot N$

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) = 2 · 1000 · 2 = **4000 I/Os**

Sort cost (S) = 2 · 500 · 2 = **2000 I/Os**

Merge cost = 1000 + 500 = **1500 I/Os**

Total cost = 4000 + 2000 + 1500 = **7500 I/Os**

At 0.1ms per I/O, total time ≈ **0.75 seconds**

Optimisation: we could combine the merge phase of sorting with the merge phase of the join, thus eliminating one scan of R and S

When is Sort-Merge Join Useful?

One or both tables are already sorted on the join key

Output must be sorted on join key (e.g., ORDER BY clause)

Typically used for **equi-joins only**

Achieves highly sequential access

- Weapon of choice for very large datasets

Hash Join

If tuple $r \in R$ and 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 partition r_i and the S tuple in partition s_i

Thus, 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 build a hash table using a hash function h on the join attributes

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

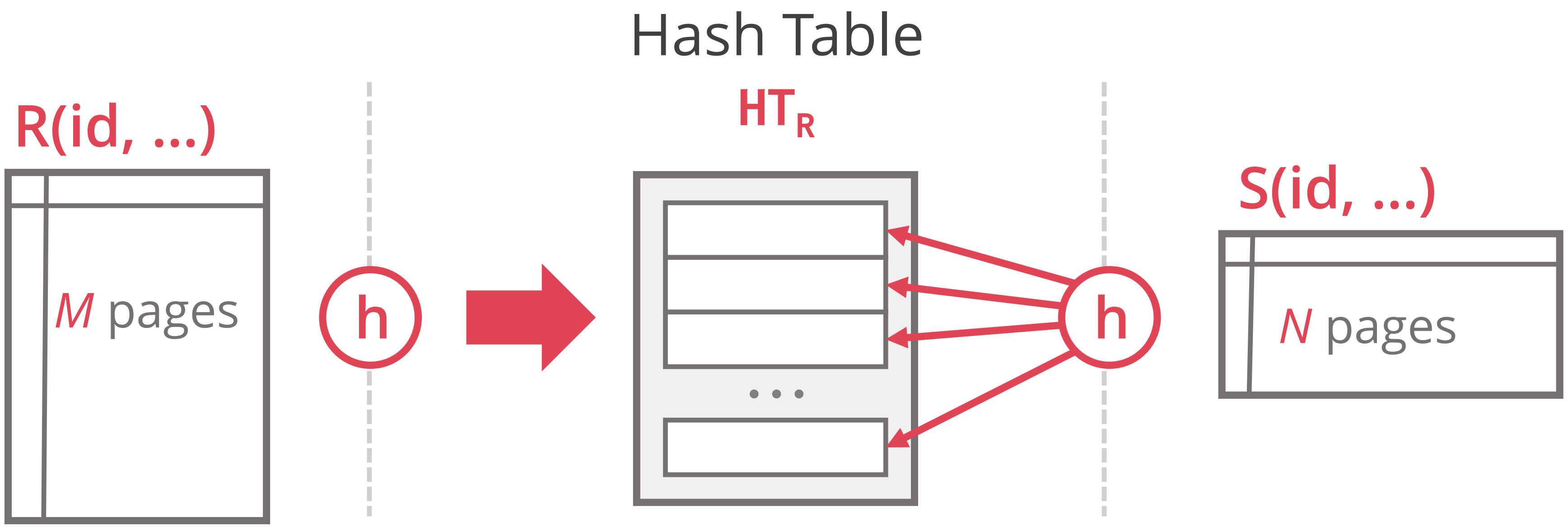
Value: full tuple or tuple identifier (used in column stores)

Phase #2: Probe

Scan the inner relation and use h on each tuple to jump to a location in the hash table and find matching tuples

Basic Hash Join Algorithm

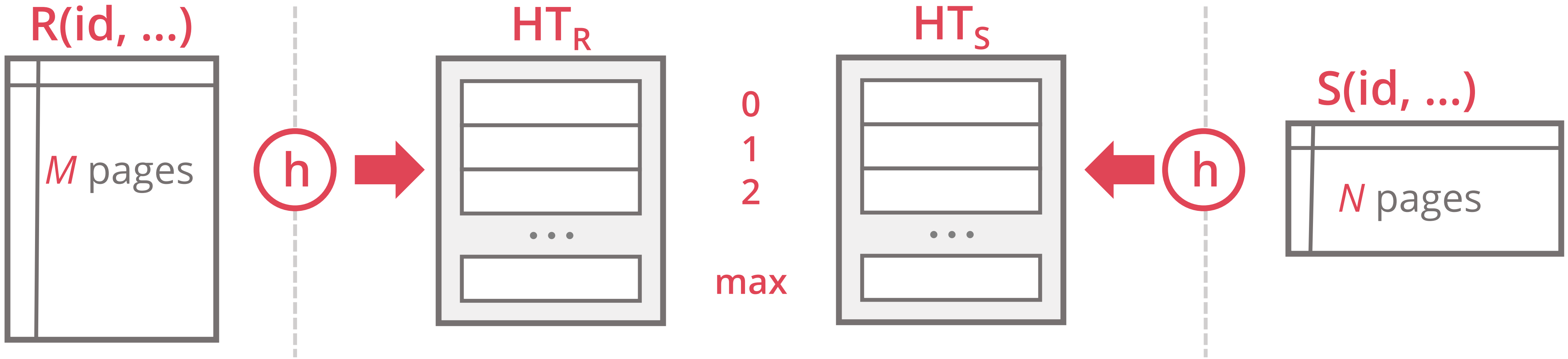
```
build hash table HT_R for R
foreach tuple s ∈ S
    emit if h(s) ∈ HT_R
```



GRACE HASH JOIN

Hash join when tables do not fit in memory

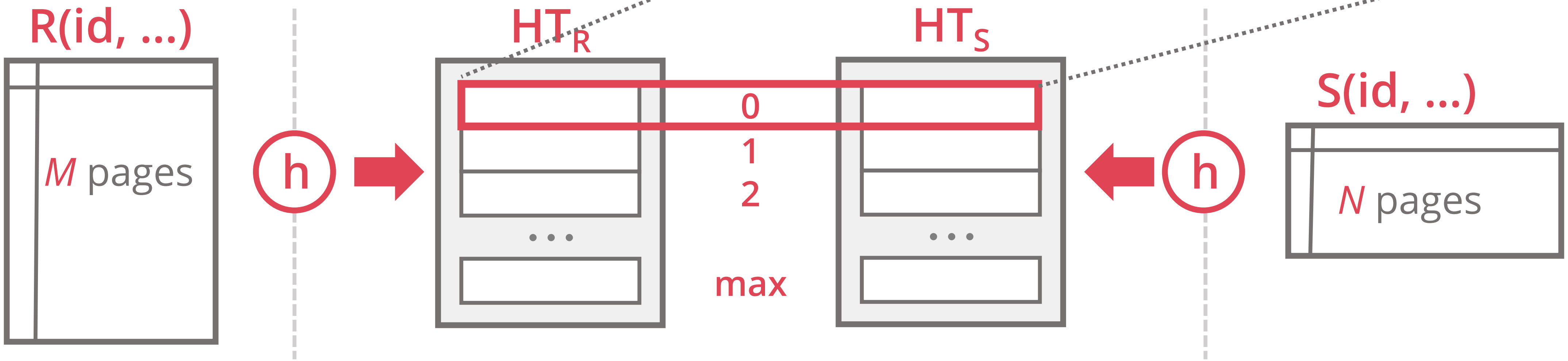
- **Build Phase**: Hash both tables on the join attribute using a hash function **h**
- **Probe Phase**: Compare tuples in corresponding partitions for each table



GRACE HASH JOIN

Join each pair of matching buckets between R and S

```
foreach tuple r ∈ bucketR,0:
  foreach tuple s ∈ bucketS,0:
    emit if r and s match
```



Grace Hash Join

If buckets do not fit in memory, use **recursive partitioning** with hash function **h_2** (≠ **h**) to split the buckets into chunks that will fit

In common cases, we have enough buffers to fit each pair of buckets

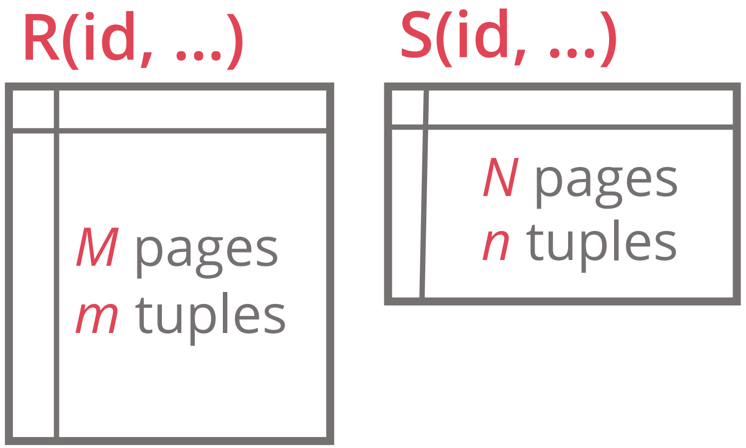
Build Phase:

Read + write both tables = $2(M + N)$ I/Os

Probe Phase:

Read both tables = $M + N$ I/Os

Total cost: $3(M + N)$



Grace Hash Join

Example database:

$M = 1000$, $m = 100{,}000$

$N = 500$, $n = 40{,}000$

Cost Analysis:

$3 \cdot (M + N) = 3 \cdot (1000 + 500) =$ **4500 I/Os**

At 0.1ms per I/O, total time ≈ **0.45 seconds**

Join Algorithms: Summary

JOIN ALGORITHM	I/O COST	TOTAL TIME
Simple Nested Loops Join	$M + (m \cdot N)$	1.4 hours
Block Nested Loops Join (using 2 input and 1 output buffer)	$M + (M \cdot N)$	50 seconds
Block Nested Loops Join (using B memory buffers)	$M + (\lceil M / (B\text{-}2) \rceil \cdot N)$	varies
Index Nested Loops Join	$M + (m \cdot C)$	varies
Sort-Merge Join	$M + N +$ (sort cost)	0.75 seconds
Hash Join	$3\,(M + N)$	0.45 seconds
Nested Loops or Hash Join (one relation fits in memory)	$M + N$	0.15 seconds

Conclusion

Use the smaller table as the outer table in nested loops joins

BNL join is also suitable for joins with inequality conditions

- Index NL join needs a clustered B+-tree index
- Hash join and sort-merge join not applicable

Hashing is almost always better than sorting for operator execution

Caveats:

- Sorting is better on non-uniform data
- Sorting is better when results need to be sorted