THE UNIVERSITY of EDINBURGH

Advanced Database Systems

Spring 2024

Lecture #18:

Joins

R&G: Chapter 14

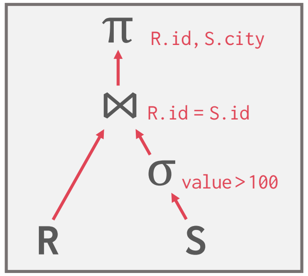
1

Join Operator

For a tuple **r ∈ R** and a tuple **s ∈ 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
```



4

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

5

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

```
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

6

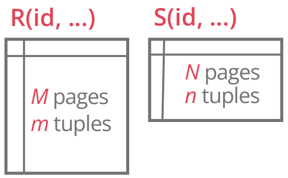
Simple Nested Loops Join

```
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
- Terrible if S does not fit in memory

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 · 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 · 1000) = **40,000,500 I/Os**
- At 0.1ms per I/O, total time ≈ **1.1 hours**

Simple Nested Loops Join

SNLJ (but with page fetches written out explicitly)

```
foreach page P_R ∈ R:
  foreach tuple r ∈ P_R:          flip loops
    foreach page P_S ∈ S:
      foreach tuple s ∈ P_S:
        emit if r and s match
```

Can we do better?

- We scan *S* for every tuple in *R*,
  - ... but we had to load an entire page of *R* into memory to get that tuple!
- Instead of finding the tuples in *S* that match a tuple in *R*,
  - ... do the check for all tuples in a page in *R* at once

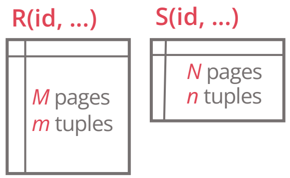
Page Nested Loops Join

```
foreach page p_R ∈ R:
  foreach page p_S ∈ S:
    foreach tuple r ∈ p_R:
      foreach tuple s ∈ p_S:
        emit if r and s match
```

This algorithm makes fewer disk accesses

- For **every page** in R, it scans S once

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



Page 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**

How many memory buffers are needed?

Just 3: one for each input, one for output

Block Nested Loops Join

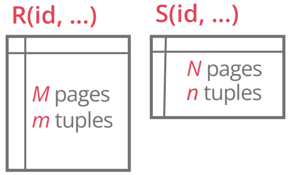
```
foreach B-2 block b_R ∈ R:
  foreach block b_S ∈ S:
    foreach tuple r ∈ b_R:
      foreach tuple s ∈ b_S:
        emit if r and s match
```

What if we have B buffers available?

B-2 buffers for scanning the outer table

1 buffer for scanning the inner table

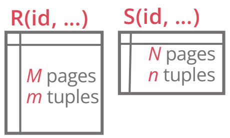
1 buffer for storing the output



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



If the outer relation (R) fits in memory ($M \leq B - 2$)

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?

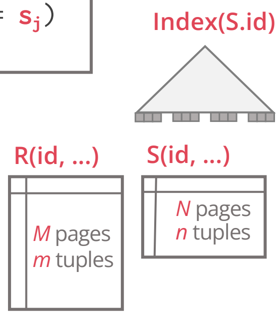
Use an index to find inner tuple matches

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
```



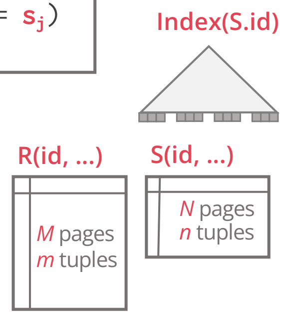
Cost: $M + m \cdot$ (cost to find all matching S tuples)

Index access cost per R tuple:

- B+ tree: 2-4 I/Os to reach a leaf + fetch matching S tuples
  - Clustered: $M + m \cdot$ (Search + # matching pages)
  - Unclustered: $M + m \cdot$ (Search + up to # matching tuples)
- Hash index: 1-2 I/Os to reach the target bucket

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 to find all matching S tuples)

The cost depends on the size of the join result

- Using an index pays off if the join is selective

Recap: 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

- Allows arbitrary join conditions

Or use an index over the inner table

- Only if matches the join condition

Sort-Merge Join

Requires equality predicate

- Equi-joins & natural joins

Phase #1: Sort

- Sort both tables on the join key(s)
  - E.g. by using the external merge sort
  - Input might already be sorted... why?

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)

19

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
```

20

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
```

21

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
```

22

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

23

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

24

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

25

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

26

27

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

27

28

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

28

29

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

29

30

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

30

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

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

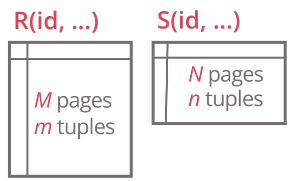
Sort Cost (S) = $2N \cdot (1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil)$ (= $2N$ · # of passes)

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

Sort-merge degenerates to simple nested-loops

Merge Cost: $M + m \cdot N$ (very unlikely!)



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**

Sort-Merge Join Refinement

Combine the last pass of merge-sort with the merge phase of join

Possible when the sum of # of runs in R and S in the penultimate (second-to-last) merge pass of sorting is at most B - 1

Example for 2-pass sort-merge join

- Read R and write out sorted runs (pass 0)
- Read S and write out sorted runs (pass 0)
- Merge R-runs and S-runs, while finding R ⋈ S matches
- Total cost = 2M + 2N + (M + N) = 2000 + 1000 + 1500 = **4500 I/Os**

Eliminates one full read and write 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

Basic In-Memory Hash Join

Requires equality predicate

Phase #1: Build

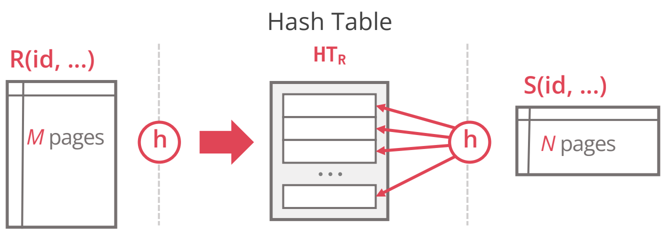
- Scan the outer relation and build a hash table using a hash function **h** on join attributes
- **Key**: the attribute(s) that the query is joining the tables on
- **Value:** full tuple or tuple identifier (used in column stores)

Phase #2: Probe

- Scan the inner relation and use **h** on each tuple to jump to a location in the hash table
- Find matching tuples there

Basic In-Memory Hash Join

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



Hash Join

What if both relations cannot fit in memory?

Idea: Decompose into smaller "partial joins"

- If tuple $r \in R$ and tuple $s \in S$ satisfy the equi-join condition, then they have the same value for the join attributes
- If that value is hashed to some value i, tuple r has to be in partition R_i and tuple s in partition S_i
- Thus, R-tuples in R_i need only to be compared with S-tuples in S_i

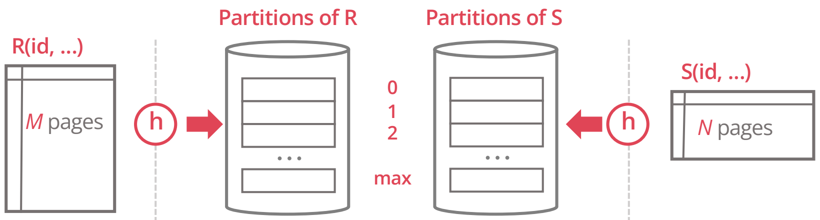
Grace Hash Join

Phase #1: Partition

Partition tuples from R and S on join attribute using a hash function h

Store partitions of R and S on scratch disk

All tuples for a given join key in same partition

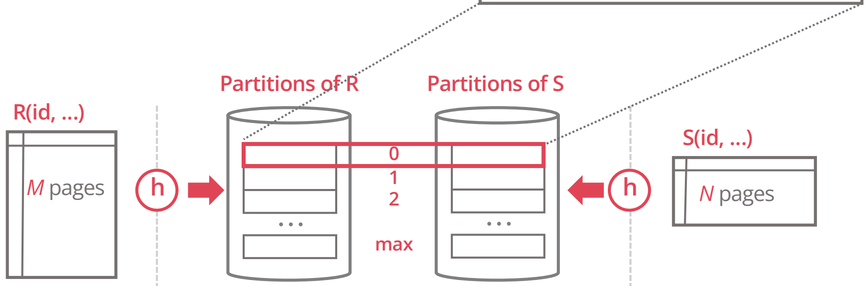


Grace Hash Join

Phase #2: Build & Probe

Join each pair of matching partitions between R and S

```
build hash table HT_R,0 for R_0
foreach tuple s ∈ S_0:
    emit if h(s) ∈ HT_R,0
```



Grace Hash Join

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

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

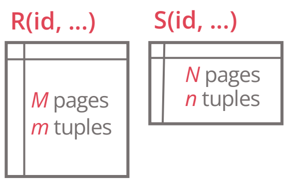
Partition Phase

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

Build & 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**

Hash Join vs. Sort-Merge Join

Sorting pros:

- Good if input already sorted, or need output sorted
- Not sensitive to data skew or bad hash functions

Hashing pros:

- For join: # of passes depends on size of smaller relation
  - E.g. if smaller relation is < **B**, basic in-memory hashing is great
- Good if input already hashed, or need output hashed

Join Algorithms: Summary

JOIN ALGORITHM	I/O COST	TOTAL TIME
Simple Nested Loops Join	$M + (m \cdot N)$	1.4 hours
Page 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	$M + (m \cdot$ access cost)	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

Summary

Nested Loops

- Works for arbitrary join condition
- Make sure to utilize memory in blocks
- Use the smaller table as the outer table

Index Nested Loops

- When you already have an index on one side
- For equi-joins mostly
- For inequality joins needs a (clustered) B+-tree index

Sort/Hash

- For equi-joins only, no index required
- Hashing better if one relation is much smaller than other
- Sorting better on non-uniform data & when results need to be sorted

No clear winners – may want to implement them all

Be sure you know the cost model for each. You will need it for query optimization!