

# Lecture 12: Joins Part I

# What you will learn about in this section

1. RECAP: Joins
2. Nested Loop Join (NLJ)
3. Block Nested Loop Join (BNLJ)
4. Index Nested Loop Join (INLJ)

# 1. RECAP: Joins

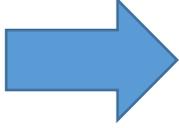
# Joins: Example

 $R \bowtie S$ 

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples  $r \in R, s \in S$  such that  $r.A = s.A$

R			S	
A	B	C	A	D
1	0	1	3	7
2	3	4	2	2
2	5	2	2	3
3	1	1		



A	B	C	D
2	3	4	2

# Joins: Example

 $R \bowtie S$ 

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples  $r \in R, s \in S$  such that  $r.A = s.A$

R			S					
A	B	C	A	D	A	B	C	D
1	0	1	3	7	2	3	4	2
2	3	4	2	2	2	3	4	3
2	5	2	2	3				
3	1	1						

# Joins: Example

 $R \bowtie S$ 

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples  $r \in R, s \in S$  such that  $r.A = s.A$

R			S					
A	B	C	A	D	A	B	C	D
1	0	1	3	7	2	3	4	2
2	3	4	2	2	2	3	4	3
2	5	2	2	3	2	5	2	2
3	1	1						

# Joins: Example

 $R \bowtie S$ 

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples  $r \in R, s \in S$  such that  $r.A = s.A$

R			S					
A	B	C	A	D	A	B	C	D
1	0	1	3	7	2	3	4	2
2	3	4	2	2	2	3	4	3
2	5	2	2	3	2	5	2	2
3	1	1			2	5	2	3

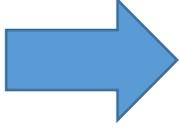
# Joins: Example

 $R \bowtie S$ 

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples  $r \in R, s \in S$  such that  $r.A = s.A$

R			S	
A	B	C	A	D
1	0	1	3	7
2	3	4	2	2
2	5	2	2	3
3	1	1		



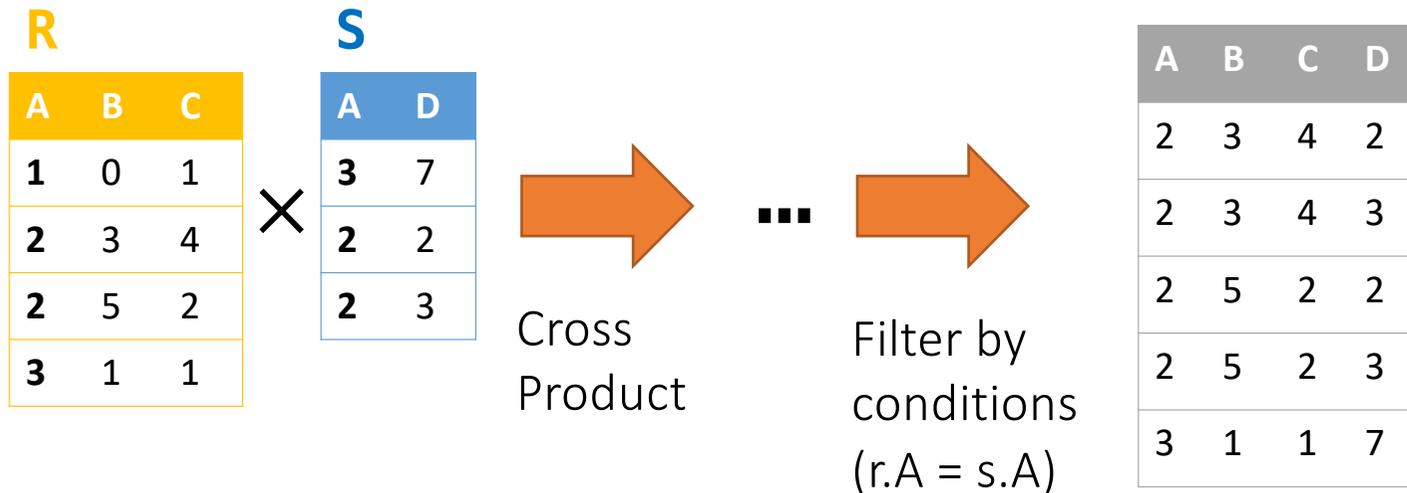
A	B	C	D
2	3	4	2
2	3	4	3
2	5	2	2
2	5	2	3
3	1	1	7

# Semantically: A Subset of the Cross Product

 $R \bowtie S$ 

```
SELECT R.A, B, C, D
FROM   R, S
WHERE  R.A = S.A
```

Example: Returns all pairs of tuples  $r \in R, s \in S$  such that  $r.A = s.A$



Can we actually implement a join in this way?

# Notes

- We write  $R \bowtie S$  to mean *join R and S by returning all tuple pairs where **all shared attributes** are equal*
- We write  $R \bowtie S \text{ on } A$  to mean *join R and S by returning all tuple pairs where **attribute(s) A** are equal*
- For simplicity, we'll consider joins on **two tables** and with **equality constraints** (“equijoins”)

However joins *can* merge > 2 tables, and some algorithms do support non-equality constraints!

## 2. Nested Loop Joins

# Notes

- We are again considering “IO aware” algorithms:  
***care about disk IO***
- Given a relation R, let:
  - $T(R)$  = # of tuples in R
  - $P(R)$  = # of pages in R
- Note also that we omit ceilings in calculations...  
good exercise to put back in!

Recall that we read / write  
entire pages with disk IO

## Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

# Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$P(R)$

**1. Loop over the tuples in  $R$**

Note that our IO cost is based on the number of *pages* loaded, not the number of tuples!

## Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S)$$

1. Loop over the tuples in  $R$
2. **For every tuple in  $R$ , loop over all the tuples in  $S$**

Have to read *all of  $S$*  from disk for *every tuple in  $R$ !*

## Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S)$$

1. Loop over the tuples in  $R$
2. For every tuple in  $R$ , loop over all the tuples in  $S$
3. **Check against join conditions**

Note that NLJ can handle things other than equality constraints... just check in the *if* statement!

## Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

What would *OUT* be if our join condition is trivial (if *TRUE*)?

*OUT* could be bigger than  $P(R)*P(S)$ ... but usually not that bad

Cost:

$$P(R) + T(R)*P(S) + OUT$$

1. Loop over the tuples in  $R$
2. For every tuple in  $R$ , loop over all the tuples in  $S$
3. Check against join conditions
4. **Write out (to page, then when page full, to disk)**

## Nested Loop Join (NLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  for  $r$  in  $R$ :  
    for  $s$  in  $S$ :  
      if  $r[A] == s[A]$ :  
        yield  $(r, s)$ 
```

Cost:

$$P(R) + T(R) * P(S) + \text{OUT}$$

*What if  $R$  ("outer") and  $S$  ("inner") switched?*



$$P(S) + T(S) * P(R) + \text{OUT}$$

Outer vs. inner selection makes a huge difference-  
DBMS needs to know which relation is smaller!

# 3. IO-Aware Approach: Block Nested Loop Join

# Block Nested Loop Join (BNLJ)

Given  $B+1$  pages of memory

Cost:

Compute  $R \bowtie S$  on  $A$ :

for each  $B-1$  pages  $pr$  of  $R$ :

for page  $ps$  of  $S$ :

for each tuple  $r$  in  $pr$ :

for each tuple  $s$  in  $ps$ :

if  $r[A] == s[A]$ :

yield  $(r,s)$

$P(R)$

1. Load in  $B-1$  pages of  $R$  at a time (leaving 1 page each free for  $S$  & output)

*Note: There could be some speedup here due to the fact that we're reading in multiple pages sequentially however we'll ignore this here!*

# Block Nested Loop Join (BNLJ)

Given  $B+1$  pages of memory

```

Compute  $R \bowtie S$  on  $A$ :
  for each  $B-1$  pages  $pr$  of  $R$ :
    for page  $ps$  of  $S$ :
      for each tuple  $r$  in  $pr$ :
        for each tuple  $s$  in  $ps$ :
          if  $r[A] == s[A]$ :
            yield  $(r,s)$ 
  
```

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in  $B-1$  pages of  $R$  at a time (leaving 1 page each free for  $S$  & output)
2. For each  $(B-1)$ -page segment of  $R$ , load each page of  $S$

Note: Faster to iterate over the *smaller* relation first!

# Block Nested Loop Join (BNLJ)

Given  $B+1$  pages of memory

Compute  $R \bowtie S$  on  $A$ :

for each  $B-1$  pages  $pr$  of  $R$ :

for page  $ps$  of  $S$ :

for each tuple  $r$  in  $pr$ :

for each tuple  $s$  in  $ps$ :

**if  $r[A] == s[A]$ :**

**yield  $(r,s)$**

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S)$$

1. Load in  $B-1$  pages of  $R$  at a time (leaving 1 page each free for  $S$  & output)
2. For each  $(B-1)$ -page segment of  $R$ , load each page of  $S$
3. **Check against the join conditions**

BNLJ can also handle non-equality constraints

# Block Nested Loop Join (BNLJ)

Given  $B+1$  pages of memory

```

Compute  $R \bowtie S$  on  $A$ :
  for each  $B-1$  pages  $pr$  of  $R$ :
    for page  $ps$  of  $S$ :
      for each tuple  $r$  in  $pr$ :
        for each tuple  $s$  in  $ps$ :
          if  $r[A] == s[A]$ :
            yield  $(r,s)$ 
  
```

Again,  $OUT$  could be bigger than  $P(R)*P(S)$ ... but usually not that bad

Cost:

$$P(R) + \frac{P(R)}{B-1} P(S) + OUT$$

1. Load in  $B-1$  pages of  $R$  at a time (leaving 1 page each free for  $S$  & output)
2. For each  $(B-1)$ -page segment of  $R$ , load each page of  $S$
3. Check against the join conditions
- 4. Write out**

# Joins, A Cage Match: BNLJ vs. NLJ

*Message: It's all about the memory!*

## BNLJ vs. NLJ: Benefits of IO Aware

- Example:

- R: 500 pages
- S: 1000 pages
- 100 tuples / page
- We have 12 pages of memory (B = 11)

*Ignoring OUT here...*

- NLJ: Cost = 500 + **50,000\*1000 = 50 Million IOs**  $\approx$  **140 hours**

- BNLJ: Cost = 500 +  $\frac{500*1000}{10}$  = **50 Thousand IOs**  $\approx$  **0.14 hours**

A very real difference from a small change in the algorithm!

## BNLJ vs. NLJ: Benefits of IO Aware

- In BNLJ, by loading larger chunks of R, we minimize the number of full *disk reads* of S
  - We only read all of S from disk for ***every (B-1)-page segment of R!***
  - Still the full cross-product, but more done only *in memory*

NLJ

$$P(R) + T(R) * P(S) + \text{OUT}$$



BNLJ

$$P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

BNLJ is faster by roughly  $\frac{(B-1)T(R)}{P(R)}$  !

## BNLJ vs. NLJ: Benefits of IO Aware

- Example:

- R: 500 pages
- S: 1000 pages
- 100 tuples / page
- We have 12 pages of memory (B = 11)

*Ignoring OUT here...*

- NLJ: Cost = 500 + **50,000\*1000 = 50 Million IOs**  $\approx$  **140 hours**

- BNLJ: Cost = 500 +  $\frac{500*1000}{10}$  = **50 Thousand IOs**  $\approx$  **0.14 hours**

A very real difference from a small change in the algorithm!

# 4. Smarter than Cross-Products: Indexed Nested Loop Join

# Smarter than Cross-Products: From Quadratic to Nearly Linear

- All joins that compute the **full cross-product** have some **quadratic** term

- For example we saw:
 
$$\text{NLJ } P(R) + T(R)P(S) + \text{OUT}$$

$$\text{BNLJ } P(R) + \frac{P(R)}{B-1} P(S) + \text{OUT}$$

- Now we'll see some (nearly) linear joins:
  - $\sim O(P(R) + P(S) + \text{OUT})$ , where again **OUT** could be quadratic but is usually better

We get this gain by *taking advantage of structure*- moving to equality constraints ("equijoin") only!

## Index Nested Loop Join (INLJ)

```
Compute  $R \bowtie S$  on  $A$ :  
  Given index  $idx$  on  $S.A$ :  
    for  $r$  in  $R$ :  
       $s$  in  $idx(r[A])$ :  
        yield  $r, s$ 
```

Cost:

$$P(R) + T(R) * L + OUT$$

where  $L$  is the IO cost to access all the distinct values in the index; assuming these fit on one page,  $L \sim 3$  is good est.

→ We can use an index (e.g. B+ Tree) to *avoid doing the full cross-product!*

# Summary

- We covered joins--an ***IO aware*** algorithm makes a big difference.
- Fundamental strategies: blocking and reorder loops (asymmetric costs in IO)
- Comparing nested loop join cost calculation is something that I will definitely ask you!