CMSC424: Database Design

Instructor: Amol Deshpande

amol@cs.umd.edu

Spring 2020 – Online Instruction Plan

- Week 1: File Organization and Indexes
- Week 2 (Reading Homework Due April 6):
  - Overview, Measures of Cost, Selections
  - Join Operation
  - Sorting, and Other Operators
- Week 3: Query Optimization; Transactions 1
- Week 4: Transactions 2
- Week 5: Parallel Database and MapReduce

Join Operation

- Book Chapters
  - 12.5
- Key topics:
  - Simplest way to do a join as a nested for loop
  - How to make it more efficient by accounting for "blocked" nature of data
  - How to use "indexes" for more efficient joins (and when they are more efficient)
  - Sorting and hashing based approaches
    - And their limitations

Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *select * from R, S where |R.a – S.a | < 0.5*
  - *Not an "equi-join"*

- Option 1: <u>Nested-loops</u>

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)
- Can be used for any join condition
  - As opposed to some algorithms we will see later
- R called *outer relation*
- S called *inner relation*

Nested-loops Join

- Cost ? Depends on the actual values of parameters, especially memory
- *b_r, b_s → Number of blocks of R and S*
- *n_r, n_s → Number of tuples of R and S*
- Case 1: Minimum memory required = 3 blocks
  - One to hold the current *R* block, one for current S block, one for the result being produced
  - Blocks transferred:
    - Must scan *R* tuples once: b_r
    - For each *R* tuple, must scan *S:* $n_r * b_s$
  - Seeks ?
    - $n_r + b_r$

Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $n_r * b_s + b_r$
  - Seeks: $n_r + b_r$
- Example:
  - Number of records -- R: $n_r = 10{,}000$, S: $n_s = 5000$
  - Number of blocks -- R: $b_r = 400$, S: $b_s = 100$
- Then:
  - blocks transferred: 10000 * 100 + 400 = 1,000,400
  - seeks: 10400
- What if we were to switch R and S ?
  - 2,000,100 block transfers, 5100 seeks
- Matters

Nested-loops Join

- Case 2: S fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2
- Example:
  - Number of records -- R: $n_r = 10{,}000$, S: $n_s = 5000$
  - Number of blocks -- R: $b_r = 400$, S: $b_s = 100$
- Then:
  - blocks transferred: 400 + 100 = 500
  - seeks: 2
- This is orders of magnitude difference

Block Nested-loops Join

- Simple modification to “nested-loops join”
  - Block at a time

for each block B_r in R

for each block B_s in S

for each tuple r in Br

for each tuple s in Bs

check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- <u>Case 1: Minimum memory required = 3 blocks</u>
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: 2 * b_r
- For the example:
  - blocks: 40400, seeks: 800

Block Nested-loops Join

- Case 1: Minimum memory required = 3 blocks
  - Blocks transferred: $b_r * b_s + b_r$
  - Seeks: $2 * b_r$
- Case 2: S fits in memory
  - Blocks transferred: $b_s + b_r$
  - Seeks: 2
- What about in between ?
  - Say there are 50 blocks, but *S* is 100 blocks
  - Why not use all the memory that we can…

Block Nested-loops Join

- <u>Case 3: 50 blocks (S = 100 blocks) ?</u>

  for each group of 48 blocks in R

    for each block B_s in S

      for each tuple r in the group of 48 blocks

        for each tuple s in Bs

          check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Why is this good ?
  - We only have to read *S* a total of $b_r/48$ times (instead of b_r times)
  - Blocks transferred: $b_r * b_s / 48 + b_r$
  - Seeks: $2 * b_r / 48$

Index Nested-loops Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- Nested-loops

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)
- Suppose there is an index on *S.a*
- *Why not use the index instead of the inner loop ?*

  for each tuple r in R

  use the index to find S tuples with S.a = r.a

Index Nested-loops Join

- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *Why not use the index instead of the inner loop ?*

  for each tuple r in R

  use the index to find S tuples with S.a = r.a

- Cost of the join:
  - $b_r\,(t_T + t_S) + n_r * c$
  - *c == the cost of index access*
    - *Computed using the formulas discussed earlier*

Index Nested-loops Join

- Restricted applicability
  - An appropriate index must exist
  - What about |*R.a* – *S.a*| < *5* ?
- Great for queries with joins and selections

*select **

from accounts, customers

where accounts.customer-SSN = customers.customer-SSN and

accounts.acct-number = "A-101"

- Only need to access one SSN from the other relation

So far…

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
  - If the smaller relation fits in memory, then cost:
    - $b_r + b_s$
    - This is the best we can hope if we have to read the relations once each
  - CPU cost of the inner loop is high
  - Typically used when the smaller relation is really small (few tuples) and index nested-loops can't be used
- Index Nested-loops join
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - select balance from customer, accounts where customer.name = "j. s." and customer.SSN = accounts.SSN

Hash Join

- <u>Case 1: Smaller relation *(S)* fits in memory</u>
- Nested-loops join:

  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a
- Cost: $b_r + b_s$ transfers, 2 seeks
- The inner loop is not exactly cheap (high CPU cost)

- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a

Hash Join

- Case 1: Smaller relation *(S)* fits in memory
- Hash join:

  read S in memory and build a hash index on it

  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a
- Cost: $b_r + b_s$ transfers, 2 seeks (unchanged)
- Why good ?
  - CPU cost is much better (even though we don't care about it too much)
  - Performs much better than nested-loops join when *S* doesn't fit in memory (next)

Hash Join

- <u>Case 2: Smaller relation *(S)* doesn't fit in memory</u>
- Two "phases"
- Phase 1:
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk
    - R gets partitioned into R1, R2, …, Rk
  - Similarly, read and partition S, and write partitions S1, S2, …, Sk to disk
  - Only requirement:
    - Each S partition fits in memory

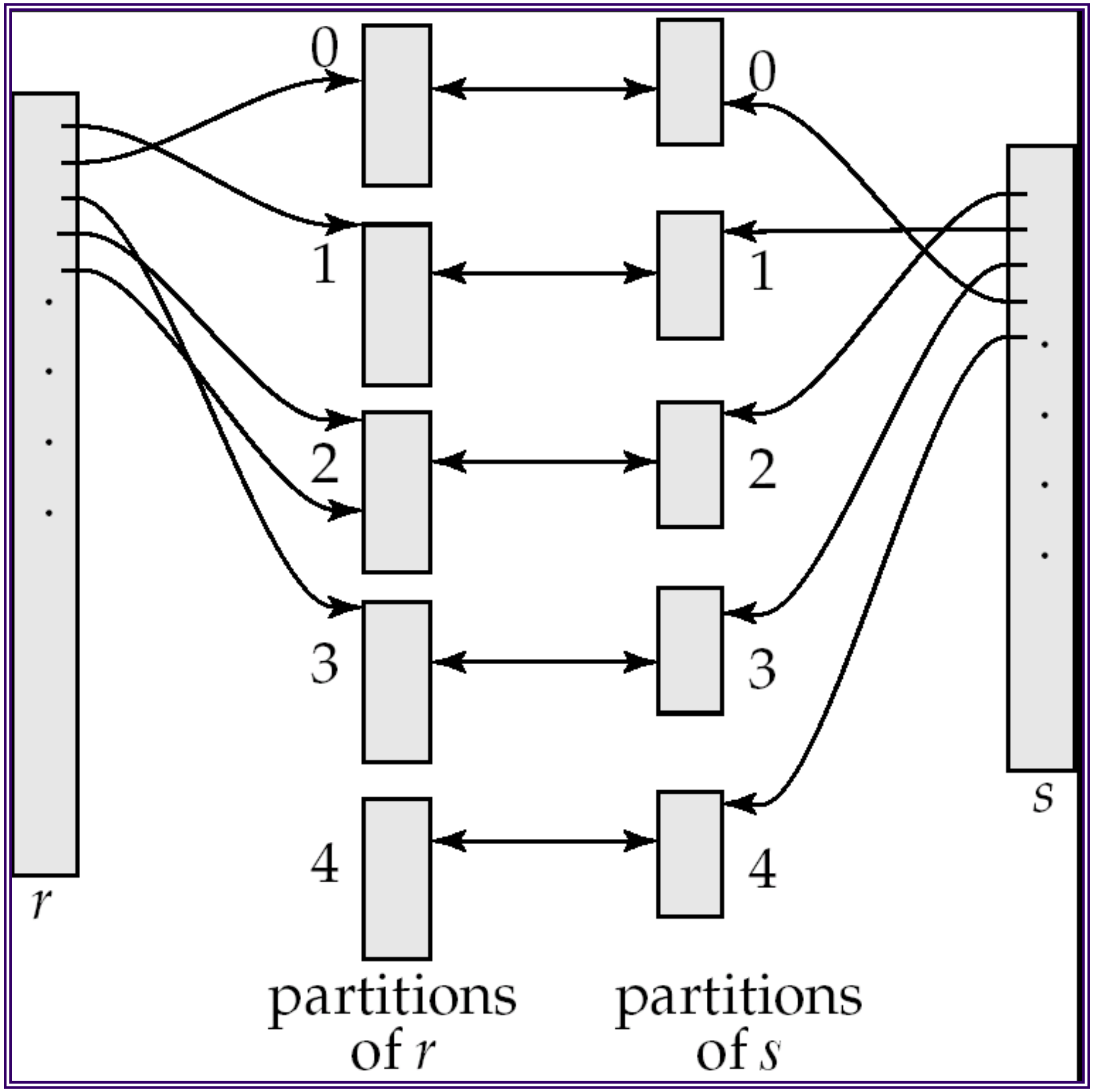
Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 2:
  - Read S1 into memory, and bulid a hash index on it (S1 fits in memory)
    - Using a different hash function, $h_2(a)$
  - Read R1 block by block, and use the hash index to find matches.
  - Repeat for S2, R2, and so on.

Hash Join

- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases":
- Phase 1:
  - Partition the relations using one hash function, $h_1(a)$
- Phase 2:
  - Read S_i into memory, and bulid a hash index on it (S_i fits in memory)
  - Read R_i block by block, and use the hash index to find matches.
- Cost ?
  - $3(b_r + b_s) + 4 * n_h$ block transfers + $2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks
    - Where b_b is the size of each output buffer
  - Much better than Nested-loops join under the same conditions

Hash Join



Hash Join: Issues

- How to guarantee that the partitions of *S* all fit in memory ?
  - Say S = 10000 blocks, Memory = M = 100 blocks
  - Use a hash function that hashes to 100 different values ?
    - Eg. *h1(a) = a % 100 ?*
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use a hash function that hashes to *100*f* different values
    - *f* is called fudge factor, typically around 1.2
    - So we may consider *h1(a) = a % 120.*
    - This is okay IF *a* is uniformly distributed

Hash Join: Issues

- Memory required ?
  - Say S = 10000 blocks, Memory = M = 100 blocks
  - So 120 different partitions
  - During phase 1:
    - Need 1 block for storing *R*
    - Need 120 blocks for storing each partition of *R*
  - So must have at least 121 blocks of memory
  - We only have 100 blocks
- Typically need *SQRT(|S| * f)* blocks of memory
- So if S is 10000 blocks, and f = 1.2, need 110 blocks of memory
- If memory = 10000 blocks = 10000 * 4 KB = 40MB (not unreasonable)
  - Then, S can be as large as *10000*10000/1.2* blocks = *333 GB*

Hash Join: Issues

- What if we don't have enough memory ?
  - *Recursive Partitioning*
  - Rarely used, but can be done

- What if the hash function turns out to be bad ?
  - We used *h1(a) = a % 100*
  - Turns out all *values of a* are multiple of 100
  - So *h1(a) is always = 0*
- Called *hash-table overflow*
- Overflow avoidance: Use a good hash function
- Overflow resolution: Repartition using a different hash function

Hybrid Hash Join

- Motivation:
  - R = 10000 blocks, S = 101 blocks, M = 100 blocks
  - So *S* doesn't fit in memory
- Phase 1:
  - Use two partitions
    - Read 10000 blocks of R, write partitions R1 and R2 to disk
    - Read 101 blocks of S, write partitions S1 and S2 to disk
  - Only need 3 blocks for this (so remaining 97 blocks are being wasted)
- Phase 2:
  - Read S1, build hash index, read R1 and probe
  - Read S2, build hash index, read R2 and probe

- Alternative:
  - Don't write partition *S1* to disk, just keep it memory – there is enough free memory for that

Hybrid Hash Join

- Motivation:
  - R = 10000 blocks, S = 101 blocks, M = 100 blocks
  - So *S* doesn’t fit in memory
- Alternative:
  - Don’t write partition *S1* to disk, just keep it memory – there is enough free memory
- Steps:
  - Use a hash function such that *S1 = 90 blocks, and S2 = 10 blocks*
  - Read S1, and partition it
    - Write S2 to disk
    - Keep S1 in memory, and build a hash table on it
  - Read R1, and partition it
    - Write R2 to disk
    - Probe using R1 directly into the hash table
  - Saves huge amounts of I/O

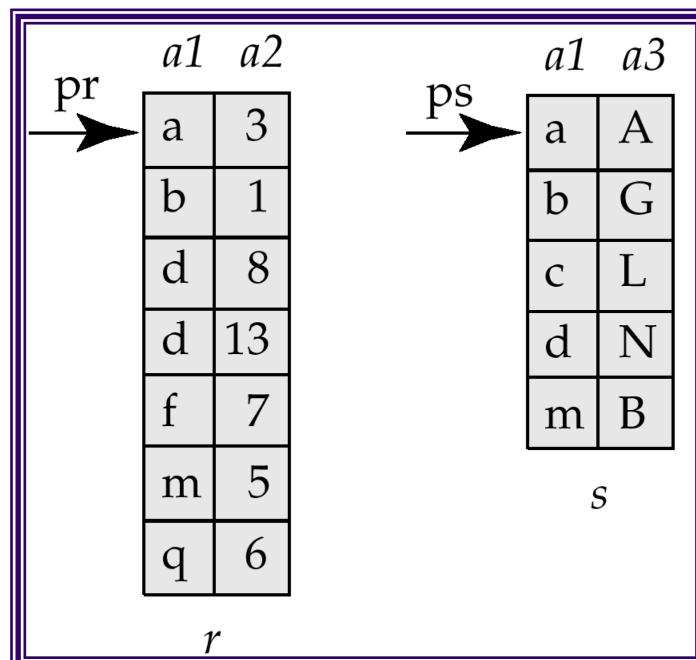
So far…

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
  - Very useful when we have selections that return small number of tuples
    - select balance from customer, accounts where customer.name = “j. s.” and customer.SSN = accounts.SSN
- Hash joins
  - Join algorithm of choice when the relations are large
  - Only applies to equi-joins (since it is hash-based)
- Hybrid hash join
  - An optimization on hash join that is always implemented

Merge-Join (Sort-merge join)

- Pre-condition:
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this algorithms
- Called “sort-merge join” sometimes

*select **
from r, s
where r.a1 = s.a1



Step:

1. Compare the tuples at pr and ps

2. Move pointers down the list

- Depending on the join condition

3. Repeat

Merge-Join (Sort-merge join)

- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted ?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on *a1*
  - This might make further operations easier to do
    - E.g. duplicate elimination

Joins: Summary

- Block Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
- Hash joins – only for equi-joins
  - Join algorithm of choice when the relations are large
- Hybrid hash join
  - An optimization on hash join that is always implemented
- Sort-merge join
  - Very commonly used – especially since relations are typically sorted
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC