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 c, a where c.name = "j. s." and c.SSN = a.SSN

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*
  - Might make further operations easier
    - E.g. duplicate elimination

Merge-Join (Sort-merge join)

- Pre-condition:
  - equi-/natural joins
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this
- Called "sort-merge join" sometimes

*SELECT **
FROM r, s
WHERE r.a1 = s.a1

p_r →

a1	*a2*
a	3
b	1
d	8
d	13
f	7
m	5
q	6

r

p_s →

a1	*a3*
a	A
b	G
c	L
d	N
m	B

s

Step:

1. Compare the tuples at p_r and p_s
2. Move pointers down the list
   - Depending on the join condition
3. Repeat

Sorting *short segue*

- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of B+-tree
    - May be enough in many cases
  - But if relation not sorted, too many random accesses
- If relation small enough…
  - Read in memory, use quicksort (qsort() in C)
- What if relation too large to fit in memory ?
  - External sort-merge

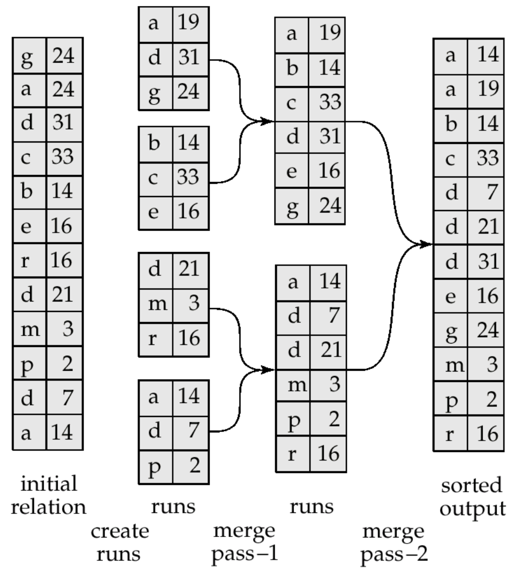
External Sort-Merge

- Divide and Conquer !!
- Let *M* denote the memory size (in blocks)
- Phase 1:
  - Read first M blocks of relation, sort, and write it to disk
  - Read the next M blocks, sort, and write to disk …
  - Say we have to do this "N" times
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*
    - *need one block per run, plus one block for output*

External sort-merge

- Phase 1:
  - Create *sorted runs of size M* each
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*
- What if *N* > *M* ?
  - Do it recursively
  - Not expected to happen
  - If *M* = 1000, can compare 1000 runs
    - (4KB blocks): can sort: 1000 runs, each of 1000 blocks, each of 4k bytes = 4GB of data

Example: External Sorting Using Sort-Merge (N >= M)



M = 3
N = 12

we assume each tuple is a block in size to simplify this example

External Merge Sort (Cont.)

Cost analysis:

- Disk for each run needs to be read and written, so:
  - $= 2b_r * (t_T + t_S)$
- Total number of merge passes required: $\lceil \log_{M-1}(b_r / M) \rceil$,
  - Each pass also reads and writes entire R
- Disk for initial run creation as well as in each pass is $2b_r$
  - for final pass, we don't count write cost
    - output may be *pipelined* (sent via memory to parent operation)

Thus total number of disk transfers for external sorting:

$$b_r (2 \lceil \log_{M-1}(b_r / M) \rceil + 1)$$

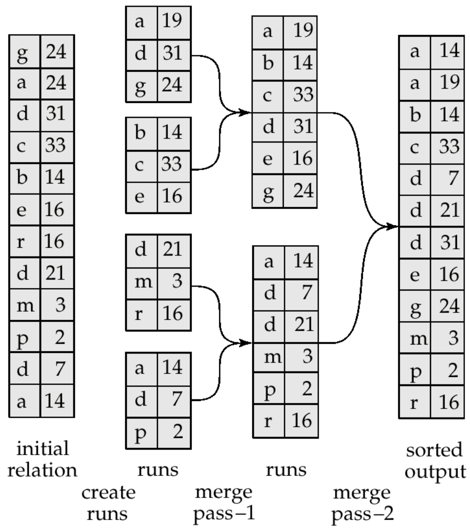
Seeks:

$$2 \lceil b_r / M \rceil + \lceil b_r / b_b \rceil (2 \lceil \log_{M-1}(b_r / M) \rceil - 1)$$

b_b is #blocks read at a time, and how many output blocks needed.

Unless otherwise specified, we assume $b_b = 1$.

Example: External Sorting Using Sort-Merge (N >= M)



M = 3
N = 12

$b_r(2\lceil \log_{M-1}(b_r / M)\rceil + 1)$ blocks

seeks:

$2\lceil b_r / M\rceil + \lceil b_r / b_b\rceil (2\lceil \log_{M-1}(b_r / M)\rceil - 1)$

- Example:
  - For $b_r = 12$, $M = 3$
  - Disk transfers = $12(2\lceil \log_2(12 / 3)\rceil + 1) = 60$
  - Seeks = $2\lceil 12/3\rceil + 12(\lceil 2 \log_2(12 / 3)\rceil - 1) = 8 + 36 = 44$

pop the stack! *segue over*

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

Case 1: Smaller relation (S) fits in memory

- *recall* 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

Case 1: Smaller relation (S) fits in memory

for each tuple r in R

for each tuple s in S

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
  - Much better than nested-loops join when *S* doesn't fit in memory (next)

Hash Join

Case 2: Smaller relation (S) doesn't fit in memory

- Basic idea:
  - partition tuples of each relation into sets that have same value on join attributes
  - *must be equi-/natural join*
- Phase 1:
  - Read R block by block and partition using a hash function:
    - $h_1(a)$ // assume has k distinct outputs
  - Create one partition for each possible value of $h_1(a)$ (k partitions)
  - Write the partitions to disk:
    - R gets partitioned into $R_1, R_2, \ldots, R_k$
  - Similarly, read and partition S, and write partitions $S_1, S_2, \ldots, S_k$ to disk
  - Requirements:
    - Room for single R block, single output block for each hash value
    - Each S partition fits into remaining memory

Hash Join

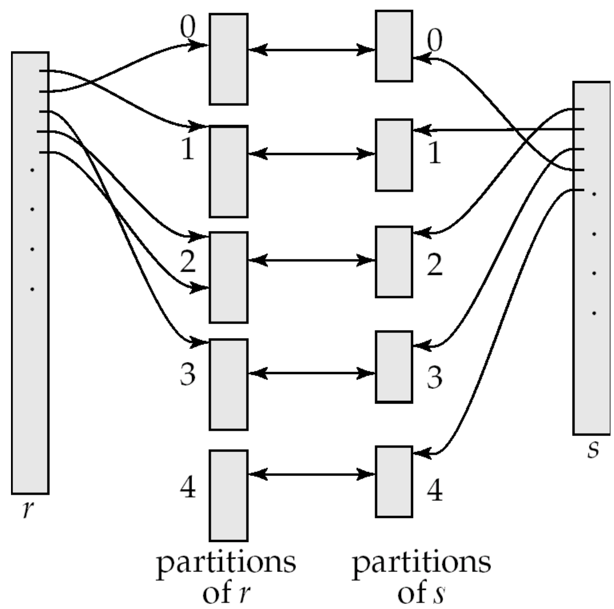
- Case 2: Smaller relation *(S)* doesn't fit in memory
  - Phase 1
  - Phase 2:
    - Read S_i into memory, and build a hash index on it (S_i fits in memory)
      - *Use a different hash function from the partition hash: $h_2(a)$*
    - Read R_i block by block, and use the hash index to find matches.
    - Repeat for all i.

Hash Join

"probe" input

"build" input

k = 5
num hash values



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 build 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 ? *remember, we ignore last output*
  - $3(b_r + b_s)$ block transfers
    - *R* or *S* might have partially full block to be read and written (ignored)
  - $+ 2(\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil)$ seeks (seek count unclear)
    - Where b_b is the size of each input buffer (p 716)
  - Much better than Nested-loops join under the same conditions

Hash Join: Issues

- How to guarantee that each partition of *S* fits in memory ?
  - Say S = 10,000 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 $h_1(a) = a \% 120$.
    - This is okay IF *a* is nearly uniformly distributed
- What if just set hash to output 200 values?
  - would need per-value output block in mem during build phase
  - oops

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
  - Need:
    - $M > n_h + 1$
    - each partition of S to fit in M-1 (why not R?)
    - space for hash build on h_2 (small, so usually ignored)
  - Example:
    - h_n = 109, average size = 10,000/109 = 91.7

Hash Join: If S_i Too Large

- Avoidance
  - Fudge factor
- Resolution
  - partition w/ a third hash: h_3
  - also partition R_i
  - go through each sub-partition
  - this approach could be used for *every* partition

Hash Join: Example

Estimate cost of single-step hash-join on *R* and *S*. *Assume:*

b_r = 2000, b_s = 1000, *M* = 202, fudge factor in this example = 1.0

Partitions of *R*?

R partition sizes do not matter. Each partition of *S* needs to fit.

During the merge phase we need 1 block for *R*, 1 for output, and then have 200 for *S*: 5 partitions for S, so 5 partitions for *R*

Block transfers for the partitioning phase?

Each block of *R* and *S* must be read and written once, so: 2 * (2000+1000) = 6000

Block transfers during the second (join) phase?

2000 +1000 = 3000 because we ignore the final writes (pipelining)

How many seeks in join phase?

We ignore the final writes, so for each set of partitions, we seek to beginning of S_i to read it into memory, then seek to beginning of R_i and go through block by block (it does not fit into memory). Total num seeks = 5(1+1) = 10.

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
- 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