Query Processing and Advanced Queries

Query Optimization (4)

Two-Pass Algorithms Based on Hashing

$$R \bowtie S$$

- If both input relations R and S are too large to be stored in the buffer, hash all the tuples of both relations applying the same hash function to the join attribute(s).
- Hash function h has domain of k hash values, i.e. k buckets.
- Only tuples from R and S that fall into the same bucket i can join.
- Hash first relation R, then relation S, write the buckets to disk.

Two-Pass Algorithms Based on Hashing

- To hash relation R, read it block by block.
- Allocate one buffer block to each of the *k* buckets.
- For each tuple t, move it to the buffer of *h*(t).
- If a buffer is full, write it to disk and initialize it.
- Finally, write to disk all partially-full buffer blocks.
- I/O cost is B(R).
- Memory requirement M = *k*+1 (k for buckets and 1 for reading tuples from R).

Two-Pass Algorithms Based on Hashing

- For each *i*, read the *i*-th bucket of R into memory, and read the *i*-th bucket of S into memory, one block at a time.
- For each tuple $s \in S$ in the buffer block, determine matching tuples $r \in R$ and output the join result (r,s).
- We assume that each bucket fits into main memory.

Two-Pass Algorithms Based on Hashing

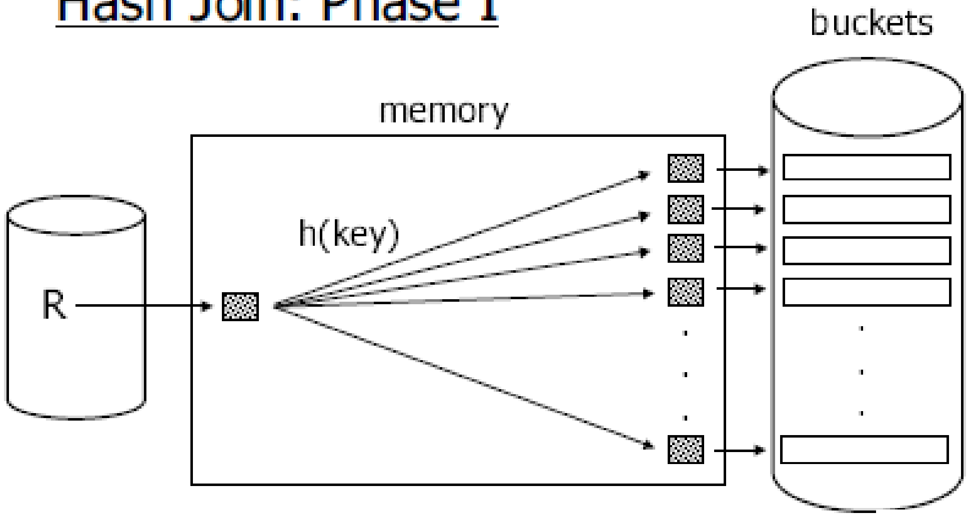
Hash join

Hash function h, range 0 . . . k
Buckets for R: G0, G1, ... Gk
Buckets for S: H0, H1, ... Hk

Algorithm
(1) Hash R tuples into G buckets
(2) Hash S tuples into H buckets
(3) For i = 0 to k do
match tuples in buckets Gi, Hi and output results

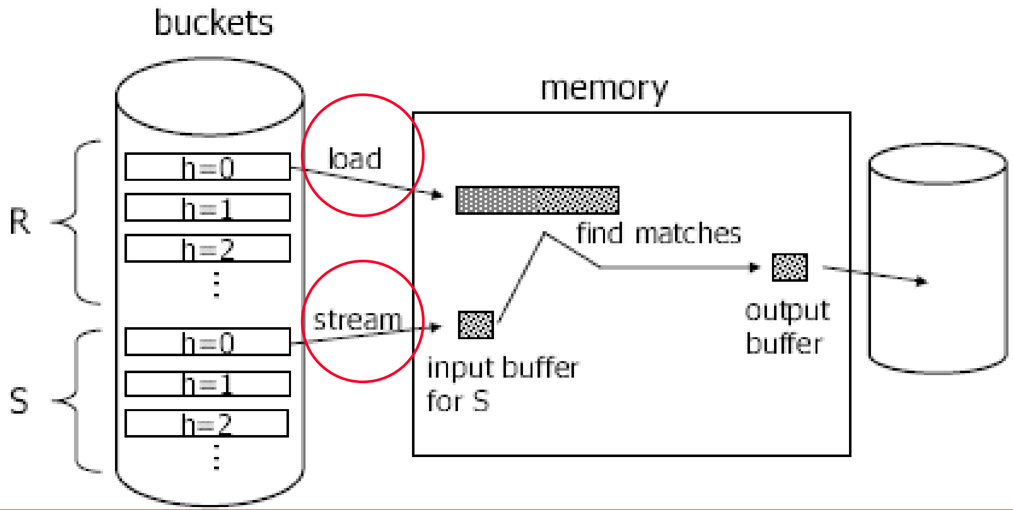
Two Phases

Hash Join: Phase I



Do the same for S

Hash Join: Phase II



Two-Pass Algorithms Based on Hashing

Example hash function: even/odd buckets

R	S
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

	R	S
Even	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

Two-Pass Algorithms Based on Hashing

- Cost
- "Bucketize:" Read R + write
  Read S + write
- Join: Read R, S

- Total cost = 3 (B(R)+B(S))
- This is an approximation, since buckets will vary in size, and we have to round up to full blocks.

Two-Pass Algorithms Based on Hashing

- Memory requirements
- Size of R bucket = B(R)/(M-1)

  k = M-1 = number of hash buckets

  - This is assuming that all hash buckets of R have the same size.
- Same calculation for S.
- The buckets for the smaller input relation must fit into main memory.

$$B(R)/(M-1) \leq M-1, \text{i.e. } M \geq \sqrt{B(R)}$$

$$B(S)/(M-1) \leq M-1, \text{i.e. } M \geq \sqrt{B(S)}$$

Index-Based Algorithms

$$R \bowtie S$$

- Index-based algorithms are especially useful for the selection operator, but also for the join operator.
- We distinguish clustering and non-clustering indexes.
- A *clustering index* is an index where all tuples with a given search key value appear on (roughly) as few blocks as possible.
- One relation can have only one clustering index, but multiple *non-clustering* indexes.

Index-Based Algorithms

Index join

For each $r \in R$ do
  $X \leftarrow$ index (S, C, r.C)
  for each $s \in X$ do
    output (r,s)

index(rel, attr, value)
  returns the set of rel tuples with attr = value

Index-Based Algorithms

- Example
  Assume S.C index exists; 2 levels.
  Assume R clustered, unordered.
  Assume S.C index fits in memory.
- Cost
  reads of R: 500 IOs
  for each R tuple:
  - probe index – no IO
  - if match, read S tuple: 1 IO.

Index-Based Algorithms

What is expected number of matching tuples?

(a) say S.C is key, R.C is foreign key
then expect 1 match

(b) say V(S,C) = 5000, T(S) = 10,000
with uniform distribution assumption
expect 10,000/5,000 = 2 matching tuples.

Index-Based Algorithms

Total cost of index join

(a) Total cost = 500+5000(1)1 = 5,500 IO

(b) Total cost = 500+5000(2)1 = 10,500 IO

Index-Based Algorithms

What if index does not fit in memory?

Example: say S.C index is 201 blocks.
(1 root node, and 200 leaf nodes)

- Keep root + 99 leaf nodes in memory.
- Expected cost of each probe is

$$E = (0)\frac{99}{200} + (1)\frac{101}{200} \approx 0.5.$$

Summary of Join Algorithms

- Nested-loop join is suitable for "small" relations (relative to memory size).
- Hash-join usually is best for equi-join (join condition is equal), where relations not sorted and no indexes exist.
- Sort-merge join is good for non-equi-join e.g., R.C > S.C.
- If relations already sorted, use merge join.
- If index exists, index-join can be efficient (depends on expected result size).

Summary: Query Processing