Query Evaluation: Joins and Beyond

Major named algorithms

Block nested loop join
checks all pairs in RXS
# blocks read = |R| + ((|R|/(F-2)))|S|

Index nested loop join
index on S with join attribute as search key
# blocks read = |R| + Σ chunks (Σ distinct values x of join attribute in chunk
index cost to first block of records with S.A=x
+ # additional blocks of such records)
best: = |R| + constant*(# distinct values of A in R)
worst: (secondary index) = |R| +n_S(index cost to first block) + n_S

Merge join
• Given R and S sorted on join attribute A
• same alg. as merging sorted lists except when find equal values of R.A and S.A, output all such R,S pairs of records
# blocks read = |R| + |S| + cost to re-read of portion of S when one value of x crosses block boundaries in R
≡ |R| + |S| + Σ values, x, of A shared by tuples in R and S
((# blocks of R with records having R.A=x) -1)
* (# blocks of S with records having S.A=x)
best: = |R| + |S|
worst: = |R|+|R|^*|S| use more buffer to improve

External Sorting of file R on attribute A

• Phase 1:
  • read R into buffer F blocks at a time
  • for each buffer-full sort and write out run of size F blocks to disk
  • at end of phase 1: have [ |R|/F ] sorted runs of size F
    – remainder may be smaller
• Phase 2:
  L_0 = { runs at end of phase 1}
while [L] >1
merge groups of |F|-1 runs in L into larger runs
using (|F|-1)-way list merge: 1 input block per run
– remainder may merge fewer
L_i+1 = { newly produced runs} / [L_i+1] = [L_i]/(|F|-1)

# blocks read/written in external sort

• Phase 1 costs 2|R| for read and write
• Phase 2:
  – # times through while loop ≤ [ log_{F-1} ( |R|/F ) ]
    • tree with fanout F-1 and [ |R|/F ] leaves
  – read and write |R| blocks each time
  • rearranging records in buffer
  • repacking into blocks
  – total cost ≤ 2 |R|^* log_{F-1} ( |R|/F )
• total # block reads/writes
  ≤ 2*R ( 1 + [ log_{F-1} ( |R|/F ) ] )
• if F-1 ≥ √ |R| reduces to 4|R|
Major named algorithms, cont. 2

• Sort merge join
  – sort R and S
  – use merge join

• cost if not multiple blocks of duplicates to join:
  \[2|R| (1 + \log_{F-1}(|R|/F)) + 2|S| (1 + \log_{F-1}(|S|/F)) + |R| + |S|\]
  \[\Rightarrow \text{cost if } F \geq \max(\sqrt{|R|}, \sqrt{|S|}): \approx 5(|R| + |S|)\]

Hash join continued

• if each bucket of R contains \(\leq F-2\) blocks:
  - read in entire bucket to buffer
  - for each block of S in corresponding bucket
    - read block into buffer
    - compare records in block with all records in bucket of R
    - write resulting records of join to output block of buffer
  - can reverse roles of R and S
  - cost: \(2(|R|+|S|)\) to build hash buckets
    + \(|R|+|S|\) to read in corresponding buckets

Hash join cost

• If have family of hash functions \(h_i\) that distribute uniformly, then need at most \(i = \lceil \log_{F-1}(|R|) \rceil\) to partition \(|R|\) down to 1 block buckets.
• Analogous for S.
• Then average recursive depth is \(\log_{F-1}(\min(|R|, |S|))\)
• Then # blocks read/write \(\leq 2^\lceil \log_{F-1}(\min(|R|, |S|)) \rceil (|R|+|S|)\) to do partitioning
  + \(|R|+|S|\) to do all join calculations
• Can fail to avoid large buckets - collisions

Final named algorithm we’ll examine

• Hash join
  – if can sort R and S to get faster join, why not build hashes of R and S?
  – choose hash function \(h\) that maps values of attribute A into F-1 values
    - not pre-existing hash index
  – partition each of R, S separately using \(h\):
    - read in one block at a time
    - F-1 blocks for output, one for each hash value
    - move each record \(r\) of R to output block for \(h(r[A])\)
    - when full, write an output block to disk and link to last block output for that hash value

Sort merge versus hash

+ hash: only need to recursively partition hash buckets until fit in F-2 blocks
  - Sort merge must really use \(\lceil \log_{F-1}(|R|/F) \rceil\) and \(\lceil \log_{F-1}(|S|/F) \rceil\) levels to merge runs
+ hash: if \(\min(|R|, |S|) < (F-1)(F-2)\) and \(h_i\)’s spread values well, get read/write cost \(3(|R|+|S|)\)
- Sort merge: need \(\max(|R|, |S|) < (F-1)^2\) and no value of A for which both R and S have multiple blocks to get read/write cost \(5(|R|+|S|)\)

But sort-merge join gives sorted result; may be useful!
Observations
• general strategy: reduce to comparing records in small subsets that fit in memory
• techniques can generalize to varying degrees from equality on single shared attribute

Query Evaluation:
Beyond Joining

Selection
• Operating on only one relation (file)
• Worst case: sequential search
  – Linear time
  – Often best case too
• If have index on R.A?
  – Equality condition on R.A
    => look up cost of index
  – Range lb ≤ R.A ≤ ub condition and tree index
    => look up cost of index

Selection with multiple conditions
\( R_x = a \text{ AND } (R_y = b \text{ OR } R_z < c) \) …

• Linear search: check Boolean expression of all conditions at once
  – No extra cost – all in main memory
• If have indexes on attributes in selection
  – AND of conditions:
    • use index giving lowest cost to retrieve candidates satisfying condition on attribute of index
    – Cost to retrieve record?
    – Number of records retrieve?
    • Check other conditions on retrieved records

Selection with multiple conditions continued
• If have indexes on attributes in selection
  – OR of conditions:
    1. Retrieve records satisfying each condition using index
    2. Union retrieved sets to form result of OR
      • Total cost of Step 1 must be less than one linear scan
      • If any attribute used in condition has no index
        => only do scan

Selection with multiple conditions AND indexes giving record pointers*

If index for every attribute involved => alternative algorithm:
1. For each equality or inequality condition
   Retrieve using index, the pointers (record IDs) for records satisfying condition
2. Sort sets of pointers
3. Merge sets of pointers
   • For AND, take intersection
   • For OR, take union
4. Retrieve actual data records using pointers
   Must evaluate if will be cheaper than getting data records earlier in process

* i.e. secondary indexes
Using record pointers

- If can get pointers for all records in query result can look up data records once
- **Manipulate pointers** of candidate records
  - Smaller size
- When ready to retrieve data records
  - Sort disk block location of pointers
  - Result may be much smaller than relation
  - Read each disk block once
  - Read disk blocks contiguously

Projection

- Must read all records — linear scan
- Only issue is *duplicate removal*
  1. Most common technique: **Sort**
     - Can eliminate unwanted attributes in Stage 1 of sort
     - Shrinks record size => less blocks to write (maybe)
     - Can eliminate duplicates in merge phases of sort
  2. Alternate technique: analogous to hash-join
     1. Drop attributes don’t want and hash into F-1 buckets
     2. For each bucket
        - If bucket fits in F-1 buffer blocks, eliminate duplicates
        - Otherwise, recurse
  3. Gift: sorted file on multi-attribute sort key and attributes want are a prefix
     - When eliminate unwanted attributes, duplicates adjacent