COS 597A:
Principles of
Database and Information Systems

Query Evaluation

Algorithms and Costs

- use what learned about
  - file organizations
  - indexes
- examine relational algebra operations
- abstraction
  - relational database level – operations
  - file organization and index level } costs
  - disk organization level } costs

Issues to consider

- Read disk pages to main memory buffer pages
  => how may buffer pages F?
- What file organization – e.g. sorted?
- What Hash and Tree indexes available?
  - on what search key?
  - on what file organization?
- Buffer use by algorithm?
  - "read x pages of relation R"
    => must be enough buffer space
  - "for record of R" => record must be in buffer
    => page of R containing record must be in buffer

How execute 1 relational operation?

- Start with JOIN with condition one = field
  - R ◊◊$_{R.f=S.f}$ S
  - "meatiest"
- other JOINs, other binary operations, other unary operations just variations
- Cost counts disk page I/O
  = # I/Os to write output file (result)
  + rest of I/O cost

always size of result in pages: IGNORE

Summary of join algorithms

- Focused on join of R and S on one shared "join attribute" f : R.f=S.f
- Developed several algorithms on the board for various situations
  - what are file organizations of R & S?
  - what indexes on R & S?
- Each algorithm checks pairs of records, one from R one from S to compute R ◊◊ S
- parameters

F - number pages in buffer
M- number pages in R　　N- number pages in S
n_R - number records in R　　n_S - number records in S

Major named algorithms

Block nested loop join
checks all pairs in RXS
pages read = M+ (M/(F-2))*N

- read R F-2 pages at a time
- for each "chunk" of F-2 pages of R,
  - read S

Index nested loop join
index on S with join attribute as search key

- read R, F-2 pages at a time
- for each "chunk" of F-2 pages of R,
  - for each value of f in the chunk
    - look up matching records of S

pages read =
M + Σ_{chunks} ($\Sigma_{\text{distinct values } x_i \text{ of join attribute in chunk}}$ (
index cost to first page of records with S.f=x_i
+ # additional pages of such records))

best: ≈ M+ constant*(# distinct values of f in R)
worst (secondary index): ≈ M +n_R(index cost to first page) + n_S

Major named algorithms, cont.

Merge join

- Given R and S sorted on join attribute f
- same alg. as merging sorted lists except when find equal values of R.f and S.f, output all such R,S pairs of records as joined records

pages read = M+ N+ cost to ***re-read*** of portion of S when one value of x_i crosses page boundaries in R

= M+ N+ $\sum$ values, x_i, of f shared by tuples in R and S (((# pages of R with records having R.f = x_i) **-1**) * (# pages of S with records having S.f = x_i))

best: = M + N

worst: = M+M*N   use more buffer to improve

External Sorting of file R on attribute f

- Phase 1:
  - read R into buffer F pages at at time
  - for each buffer-full
    sort and write out *run* of size F pages to disk
- at end of phase 1: have ⌈M/F⌉ sorted runs of size F
  – remainder may be smaller
- Phase 2:
  L_0 = { runs at end of phase 1}
  while $|L_i|>1$
    merge groups of F-1 runs in L_i into larger runs
      using (F-1)-way list merge: 1 input page per run
        – remainder may merge fewer
    L_{i+1} = {newly produced runs}   // $|L_{i+1}| = \lceil |L_i|/(F-1) \rceil$

pages read/written in external sort

- Phase 1 costs 2M for read and write
- Phase 2:
  – # times through while loop ≤ $\lceil \log_{F-1}(\lceil M/F \rceil) \rceil$
    - tree with fanout F-1 and ⌈M/F⌉ leaves
  – read and write M pages each time
    - rearranging records in buffer
    - repacking into pages
  – total cost ≤ 2 M*$\lceil \log_{F-1}(\lceil M/F \rceil) \rceil$
- total # page reads/writes
  ≤ $2*M(1 + \lceil \log_{F-1}(\lceil M/F \rceil) \rceil)$
- if F-1 ≥ √M reduces to 4M

Major named algorithms, cont. 2

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

- cost if *not* multiple pages of duplicates to join:
  $2*M (1 + \lceil \log_{F-1}(\lceil M/F \rceil) \rceil)$
  + $2*N (1 + \lceil \log_{F-1}(\lceil N/F \rceil) \rceil)$
  + M+ N

⇒ cost if F ≥ max (√M, √N):
  ≈ 5(M + N)

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 f into F-1 values
    - ***not pre-existing*** hash index
  – partition each of R, S separately using *h:*
    - read in R one page at a time
    - F-1 pages for output, one for each hash value
    - move each record r of R to output page for *h*(r[f])
    - when full, write an output page to disk and link to last page output for that hash value

- hash join continued
  - if each bucket of R contains ≤ F-2 pages:
    for each bucket of R
      read in entire bucket to buffer
      for each page of S in corresponding bucket
        - read page into buffer
        - compare records in page with all records in bucket of R
        - write resulting records of join to output page of buffer
  - can reverse roles of R and S
  - cost: 2(M+N) to build hash buckets
    + M+N to read in corresponding buckets

- hash join still continued

if some corresponding buckets of R and S are *large*, i.e. contain > F-2 pages:

- have 2nd hash function h_2 hashing into F-1 values
- for each pair of large buckets of R and S, partition each bucket using h_2
- for each pair of resulting buckets with one having ≤ F-2 pages, calculate join
- for each pair of resulting *large* buckets, recurse with h_3

…

13

Hash join cost

- **If** have family of hash functions h_i that distribute uniformly, then need at most $i = \lceil \log_{F-1}(M) \rceil$ to partition R down to 1 page buckets.
- Analogous for S.
- Then average recursive depth is

  $\log_{F-1}(\min(M, N))$
- Then # pages read/write

  $\leq 2^*\lceil \log_{F-1}(\min(M, N)) \rceil^*(M+N|))$ to do partitioning

  $+ (M+N)$ to do all join calculations
- **Can** fail to avoid large buckets - collisions

14

Sort merge versus hash

\+ hash: only need to recursively partition buckets until fit in F-2 pages

\- Sort merge must really use $\lceil \log_{F-1} (\lceil M/F \rceil) \rceil$ and $\lceil \log_{F-1} (\lceil N/F \rceil) \rceil$ levels to merge runs

\+ hash: if **min**(M,N) < (F-1)(F-2) and h_i's spread values well, get read/write cost 3(M+N)

\- Sort merge: need **max**$(M,N) \leq (F-1)^2$ and no value of attribute f for which both R and S have multiple pages to get read/write cost 5(M+N)

But sort-merge join gives sorted result;

may be useful!

15

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

16

Query Evaluation: Beyond Joining

17

Selection

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

18

Selection with multiple conditions

R.x = a AND (R.y = b 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

19

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 must do scan

      => *only* do scan

20

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

21

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 page location of pointers
    - Result may be much smaller than relation
  - Read each disk page once
  - Read disk pages contiguously

22

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 pages 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 pages, 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

23