CMSC 424 – Database design
Lecture 17
Query processing

Mihai Pop

Admin

- Homework 3 is on the website
- Project part 1 due
- Midterm answers/results
  > 90 – A (6)
  75-90 – B (9)
  50-75 – C (6)
  < 50 – D (1)
- 10 improved, 10 got worse, 2 about the same

Wake up...skills you should have

- Find information online (e.g. what is this BibTex after all)
- Write a parser for a simple file format
- Adapt/incorporate an existing parser

- Manage your time (start early, evaluate difficulty of project)

- Pay attention in class

- Communicate

Complex Selections

- conjunction $\sigma_{\theta_1 \wedge \theta_2}$ s1 = # of tuples satisfying θ 1
  $s2$ = # of tuples satisfying θ 2

  combined SC = $s1 * s2/(n(r) * n(r))$

  assuming independence of predicates

- disjunction $\sigma_{\theta_1 \vee \theta_2}$

  combined SC = $1-(1-s1/n(r))*(1-s2/n(r))$

this 1 minus the probability of all predicates are satisfied at once (s1/n(r) + s2/n(r) – s1/n(r)*s2/n(r))- union of results

- negation $\sigma_{\neg\theta}$

  n($\sigma_{\neg\theta}(r)$) = n(r) - n($\sigma_{\theta}(r)$)

Multiple Index Selection

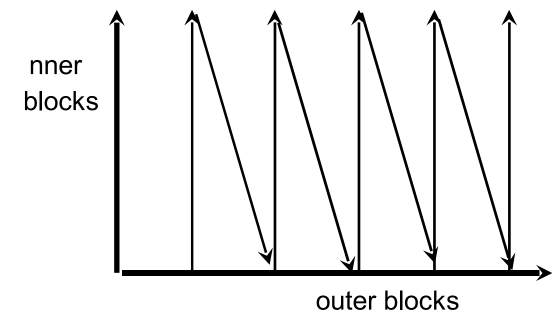
<u>GOAL:</u> apply the most restrictive one and combine multiple of them to reduce the intermediate results AS EARLY AS POSSIBLE

- conjunctive selection using one index A: select using A and then apply the remaining of the predicates on the retrieved tuple values
- conjunctive selection using a composite key index (R.A,R.B)- then create a composite key or range from the query values and search directly (range search on the first attribute only)
- conjunctive selection using two indexes A and B: search each separately and intersect the tuple identifiers (TIDs)
- disjunctive selection using two indexes A and B: search each separately and take the union of the TIDs

Join Methods: Nested Loop

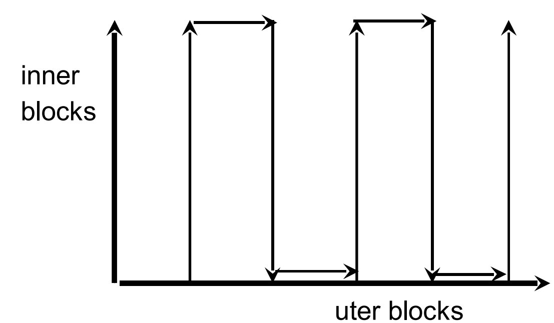
- tuple-oriented:

for each tuple t(r) in r do begin
for each tuple t(s) in s do begin
join(t(r),t(s) and append the result to the output
end
end



- block-oriented:

for each block b(r) in r do begin
for each block b(s) in s do begin
join(b(r),b(s) and append the result to the output
end
end



- reverse inner loop

similar to above but for even outer blocks we scan the inner relation in reverse

Cost of Block-Oriented Nested Loop

Buffer size M+1

- cost depends on the number of buffers and the buffer replacement strategy
  - fasten 1 block from the outer relation, M for the inner and LRU

    cost: b(r) + b(r)*b(s) assuming that b(s) > M

  - fasten M blocks from the outer relation, and 1 for the inner

  1: read M from the outer cost: M blocks

  2: for each block of s join 1 X M blocks cost: b(s) -"-

  3: repeat with the next M blocks of r until all done
  repeated b(r)/M times

  cost = [(M + b(s)]* b(r)/M = b(r)+[b(r)*b(s)]/M

- which relation should be the outer?

Join Methods: Sort-Merge-Join

- two phases
  - sorting phase: sort both relations (this can be done in parallel)
  - merging phase: join tuples during the merge

sort R on joining attribute
sort S on joining attribute
merge(sorted-R,sorted-S)

- cost with M buffers

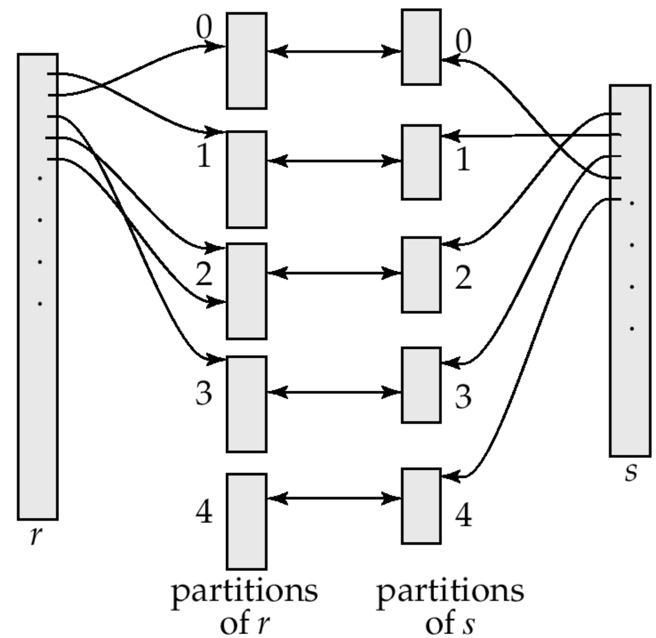
$$\text{cost} = \underbrace{b_r\,(\,2\lceil \log_{M-1}(b_r / M)\rceil + 1)}_{\text{Sorting R}} + \underbrace{b_r}_{\text{sorted WriteR}} + \underbrace{b_s\,(\,2\lceil \log_{M-1}(b_s / M)\rceil + 1)}_{\text{Sorting}} + \underbrace{b_s}_{\text{Write sorted S}} + \underbrace{b_r + b_s}_{\text{Merge}}$$

if one pass is required the expressions $\lceil \log_{M-1}(br/M)\rceil = 1$ and $\lceil \log_{M-1}(bs/M)\rceil = 1$ so the total cost is 3*b(r)+b(r)+b(r)+ 3*b(s)+b(s)+b(s) =5*b(r)+ 5*b(s)
However, if M> b(r) and b(s) then the expression evaluates to 3*b(r)+3*b(s)).

Join Methods: Hash-Join

- two phases
  - hash phase: hash both relations into hashed partitions (this can be done in parallel)
  - bucket-wise join phase: join tuples of the same partitions only

hash R on the joining into H(R) buckets
hash S on the joining into H(S) buckets
nested-loop join of corresponding buckets Hj(R),Hj(S)
or main-memory hash index join of -”-



- Number of partitions is large to make each partition of Hj(R) fit in the buffer memory
  -- each Hj(R) consists of several blocks

- We assume that buckets of Hj(R) fit in the buffer memory
  (and that after hashing the partitions of R and S have the same size with R and S):

$$\text{cost} = b(r) + b(r) + b(s) + b(s) + b(r) + b(s) = 3\,(b(r) + b(s))$$

Hash-Join Algorithm Details

The hash-join of *r* and *s* is computed as follows.

1. Partition the relation *s* using hashing function *h*. When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition *r* similarly.
3. For each *i:*
   (a) Load s_i into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one *h.*
   (b) Read the tuples in r_i from the disk one by one. For each tuple t_r locate each matching tuple t_s in s_i using the in-memory hash index. Output the concatenation of their attributes.

Relation *s* is called the **build input** and *r* is called the **probe input.**

Example of Cost of Hash-Join

- M= 20 blocks
- $b_{depositor}$= 100 *customer* $\bowtie$ *depositor*
- $b_{customer}$ = 400.
- *depositor* is the build input. Partition it into 5 partitions, each of size 20 blocks. This partitioning can be done in one pass.
- partition *customer* into 5 partitions,each of size 80. This is also done in one pass.
- Do the partition joins- for each j put 20 blocks of partition depositor(j) in memory, built the hash index, and do the probes with the 80 blocks of customer(j)
- Therefore total cost, ignoring cost of writing partially filled blocks:
  – 3(100 + 400) = 1500 I/Os

Hash-Join algorithm (Cont.)

- The value n and the hash function h is chosen such that each s_i should fit in memory.
  - Typically n is chosen as $\lceil b_s/M \rceil * f$ where f is a "fudge factor", typically around 1.2
  - The probe relation partitions r_i need not fit in memory

- Recursive partitioning required if number of partitions n is greater than number of pages M of memory.
  - instead of partitioning n ways, use $M - 1$ partitions for s
  - Further partition the $M - 1$ partitions using a different hash function
  - Use same partitioning method on r
  - Rarely required: e.g., recursive partitioning not needed for relations of 1GB or less with memory size of 2MB, with block size of 4KB.

Join Methods: Indexed-Join

- inner relation has an index (clustering or not)

```
for each  block b(r) in  r  do begin
    for each tuple t(r)   in b(r)  do begin
         search the index B on s with the value t.A of the joining attr. A
         and join(t(r),t.A)
    end
 end
```

- cost = b(r) + n(r)* cost(σ(S.A=c))
  where cost(σ(S.A=c)) is as computed for indexed selection

Estimation of Join Size: $n(R \bowtie S)$

- $0 <= n(R \bowtie S) <= n(r)*n(s)$ (0 when nothing joins and when everything joins)
  - if joining attribute is a key of R then $n(R \bowtie S) <= n(s)$
    /* each value of S.A would join to at most one value of R.A */
  - if -″- is a key of R and a foreign key of S then $n(R \bowtie S) = n(s)$
    /* each value of S.A would join to exactly one value of R.A */
  - if is not a key then
    each value of A in R appears n(s)/V(A,s) times in S, therefore,
    n(r) tuples of R produce: $n(R \bowtie S) = n(r)*n(s) / V(A,s)$

    symmetrically we can obtain: $n(R \bowtie S) = n(r)*n(s) / V(A,r)$

  if the values are different we use: $\min\{n(r)*n(s) / V(A,s) , n(r)*n(s) / V(A,r)\}$

Other Operations

- Outer Joins
  - Left outer join easy
  - Right/Full outer join (may need some bookkeeping)

- Duplicate elimination
  - Hard
  - Sort at the end and eliminate
  - Hash output and eliminate

- Aggregates
  - Sum, count, min, max easily kept during execution
  - Avg = Sum / count
  - Std = sqrt(ssum/count)

External Sorting with Sort-Merge

- external vs internal sorting: relation/file does not fit in memory

- create runs phase:

repeat until done

read M blocks of the relation (or rest if <=M)

internal sort using any sort method, e.g QuickSort(M)

write the sorted tuples into a run R data file

end

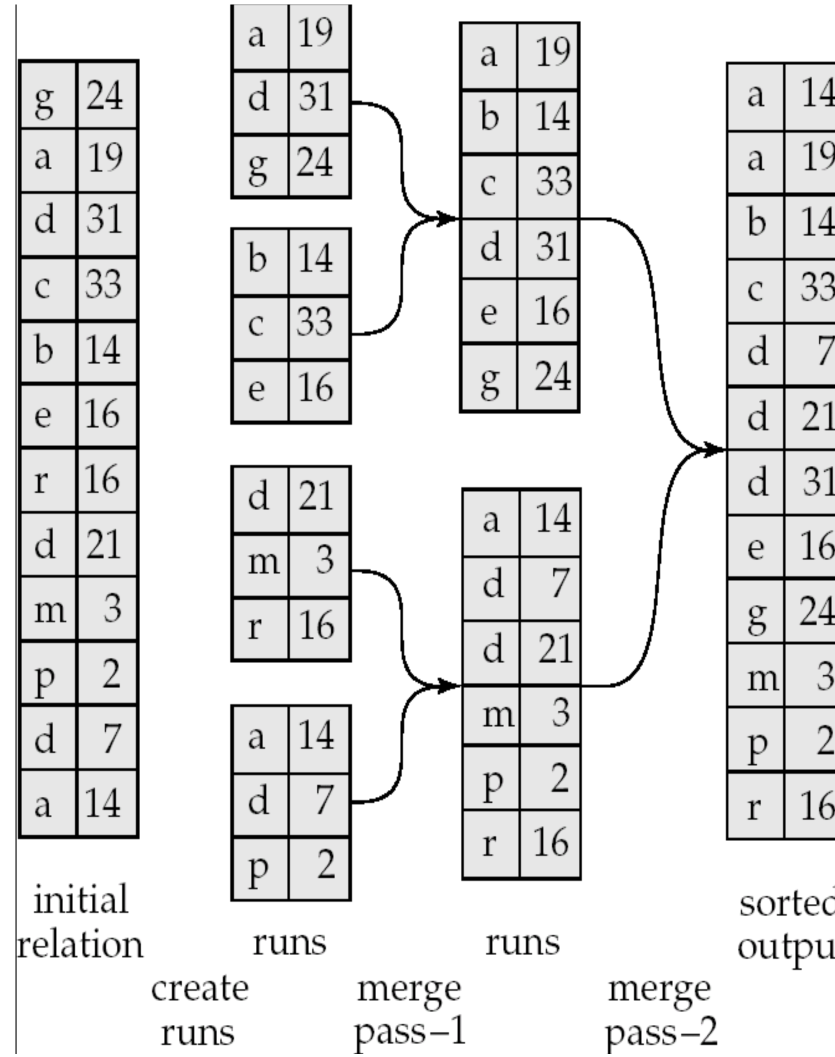
- merge-runs phase:

read one block from each run;

merge tuples on the result;

advance the pointer from the run you appended last;

if the block of a run is empty, read the next one until all blocks of all runs are done



- this assumes that a block from each run can be kept in main memory. If not, then the same algorithm has to be applied in multiple passes

External Merge Sort Cost

- Cost analysis:
  - Initial number of runs: b_r/M
  - Total number of merge passes required: $\lceil \log_{M-1}(b_r/M) \rceil$.
  - Block transfers for initial run creation is $b_r + b_r = 2b_r$
    - for final pass, we don't count write cost

      we ignore final write cost for all operations since the output of an operation may be pipelined to the display or to a parent operation without being written to disk. If pipelined, it will be counted in the cost of the follow up operator

    - Thus total number of block transfers for external sorting:

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

  - If M $\geq \lceil b_r/M \rceil$ (only one pass is required) the expression $\lceil \log_{M-1}(b_r/M) \rceil = 1$

    total cost = $3b_r$

  - However, if M > b_r then this expression evaluates to 0

    total cost =b_r ONLY