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 $\sigma_{\theta_1 \land \theta_2}$
  
  $s_1 = \# \text{ of tuples satisfying } \theta_1$
  
  $s_2 = \# \text{ of tuples satisfying } \theta_2$

  combined SC = $s_1 * s_2 / (n(r) * n(r))$

  assuming independence of predicates

• disjunction $\sigma_{\theta_1 \lor \theta_2}$

  combined SC = $1 - (1 - s_1/n(r)) * (1 - s_2/n(r))$

  this 1 minus the probability of all predicates are satisfied at once

• negation $\sigma_{\neg \theta}$

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

**GOAL:** 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

• 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
    \[ \text{cost: } b(r) + b(r) \cdot b(s) \text{ assuming that } b(s) > M \]
  – fasten M blocks from the outer relation, and 1 for the inner
  1: read M from the outer \hspace{1cm} \text{cost: } M \text{ blocks}
  2: for each block of s join 1 X M blocks \hspace{1cm} \text{cost: } b(s) \hspace{1cm} -''-
  3: repeat with the next M blocks of r until all done
    repeated \hspace{1cm} b(r)/M times
    \[ \text{cost} = \left[ (M + b(s)) \cdot \frac{b(r)}{M} \right] = b(r) + \left[ \frac{b(r) \cdot b(s)}{M} \right] \]

• 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

\[
\text{cost} = b_r \left( 2 \left\lfloor \log_{M-1} \left( \frac{b_r}{M} \right) \right\rfloor + 1 \right) + b_r + b_s \left( 2 \left\lfloor \log_{M-1} \left( \frac{b_s}{M} \right) \right\rfloor + 1 \right) + b_r + b_s
\]

if one pass is required the expressions \( \left\lfloor \log_{M-1} \left( \frac{br}{M} \right) \right\rfloor = 1 \) and \( \left\lfloor \log_{M-1} \left( \frac{bs}{M} \right) \right\rfloor = 1 \) so the total cost is \( 3b(r) + b(r) + b(r) + 3b(s) + b(s) + b(s) = 5b(r) + 5b(s) \)

However, if \( M > b(r) \) and \( b(s) \) then the expression evaluates to \( 3b(r) + 3b(s) \).
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
- $b_{\text{depositor}} = 100$
- $b_{\text{customer}} = 400$.
- $\text{depositor}$ is the build input. Partition it into 5 partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Partition $\text{customer}$ 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 $\text{depositor}(j)$ in memory, built the hash index, and do the probes with the 80 blocks of $\text{customer}(j)$
- 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 \( \lceil b_s / M \rceil \times f \) 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)

\[
\text{cost} = b(r) + n(r) \times \text{cost}\left(\sigma_{S.A=c}\right)
\]

where \(\text{cost}(\sigma_{S.A=c})\) is as computed for indexed selection.
Estimation of Join Size:\n\[ n(R \bowtie S) \]

- if joining attribute is a key of R then
  \[ n(R \bowtie S) \leq n(s) \]
  /* 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 \( \leq M \))
  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:

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

  - 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