Query Processing and Query Optimization Problem Set

12.10

a)

The number of blocks in memory buffer available for sorting $M = \frac{40 \cdot 10^6}{4 \cdot 10^3} = 10^4$. Number of blocks containing records of the given relation $br = \frac{40 \cdot 10^9}{4 \cdot 10^3} = 10^7$. Total cost of sorting C will be

$$C = \#\,of\,disk\_seeks(n_s) \times disk\_seek\_cost(t_s) + \#of\,block\_transfers(n_t) \times block\_transfer\_time(t_t)$$

$$t_s = 5 \times 10^{-3} s$$

$$t_t = \frac{4 \times 10^3}{40 \times 10^6} = 10^{-4} s$$

$$mergePass = [log_{\frac{M-1}{bb}}(\frac{br}{M})]$$

$$n_t = b_r \times (2 \times mergePass + 1)$$

$$n_s = 2 \times \frac{b_r}{M} + \frac{b_r}{b_b} \times (2 \times mergePass - 1)$$

As the equation showed above, the total amount it takes to sort the relation consist of average disk seek cost, number of disk seeks, cost of block transfer and number of block transfers.

- $bb = 1$
  - In this case, at the first run, all tuples are read in and wrote out once and there are therefore $\frac{br}{M} = 1000$ sorted runs. And this step requires $2 \times \frac{b_r}{M}$ block seeks because; $2b_r$ block transfers because every block is read in and written out.
  - In the second pass since the buffer block $b_b = 1$, $(M-1) \div b_b = 9999$ block is available for merges. So in this case, 9999 blocks are available to merge and requires $[log_{9999}1000] = 1$ mergePasses to merge all the runs. And this pass will require b_r block seeks and b_r block reads to read data from disk to memory.
  - Substitute into the equation above:

$$n_s = 2 \times 10^3 + 1 \times 10^7 \times (2 \times 1 - 1) = 1.0002 \times 10^7$$

$$C = 3 \times 10^7 \times 10^{-4} + 1.0002 \times 10^7 \times 5 \times 10^{-3} = 53010s$$

- $bb = 100$
  - In this case, at the first run, all tuples are read in and wrote out once and there are therefore $\frac{br}{M} = 1000$ sorted runs. And this step requires $2 \times \frac{b_r}{M}$ block seeks because every block is read one at a time; $2b_r$ block transfers because every block is read in and written out.

- In the second pass since the buffer block $b_b = 100$, $(M-1) \div b_b = 99.99$ block is available for merges. So in this case, only 99 blocks can be taken to merge and requires $[log_{99.99}1000] = 2$ mergePasses to merge all the runs. And this run requires $b_r \div 100 = 10^5$ disk seeks per pass because 100 blocks are read in a pass all together and it requires $3b_r$ block transfers cause every block is read in, written out and read in.

Substitute into the equation above:

$$n_s = 2 \times 10^3 + 1 \times 10^5 \times (2 \times 2 - 1) = 3.02 \times 10^5$$
$$C = 5 \times 10^7 \times 10^{-4} + 3.02 \times 10^5 \times 5 \times 10^{-3} = 6510s$$

b) As reasoned above:

The number of merge passes depends on the number of runs merged together during each pass.

- $bb = 1$

  In this case, all 1000 runs can be merged into one sorted result in one pass.

$$mergePass = [log_{\frac{M-1}{bb}}(\frac{br}{M})] = log_{9999}1000 = 1$$

- $bb = 100$

  In this case, only 99 runs can be merged into one sorted result in one pass. So two is required.

$$mergePass = [log_{\frac{M-1}{bb}}(\frac{br}{M})] = log_{99.99}1000 = 2$$

c)

- $bb = 1$

$$t_s = 1 \times 10^{-4}$$
$$new_C = 3 \times 10^7 \times 10^{-4} + 1.02 \times 10^7 \times 1 \times 10^{-4} = 4020s$$

- $bb = 100$

$$t_s = 1 \times 10^{-4}$$
$$new_C = 3 \times 10^7 \times 10^{-4} + 3.02 \times 10^5 \times 1 \times 10^{-4} = 3030.2s$$

12.11

a) $Semijoin$:

- $Semijoin$ using sorting:

  Sort both r and s first on the join attributes θ. Then perform a scan of both r and s using something similar to merge (join) algorithm and add tuples of r to the result whenever the join attributes of the current tuples of r and s match.

- $Semijoin$ using hashing:

  Create a hash index in s on the join attributes θ. Iterate thru r and for each distinct value of the join attribute, perfrom a hash lookup in s. If the hash lookup finds and returns a valid entry, add the current r tuple to result.

  - If r and s are large, they can be partitioned on the join attributes first, and then execute the above procedures applied on each partition.
  - If r is small but s is large, a hash index can be built on r and probed using s. If a s tuple matches a r tuple, the r tuple can be output and deleted from hash index.

b) $Anti-Semijoin$:

- $Anti-Semijoin$ using sorting:

  Sort both r and s first on the join attributes θ. Then perform a scan of both r and s using something similar to merge (join) algorithm and add tuples of r to the result if no tuple of s satisfies θ for corresponding r tuple.

- $Anti-Semijoin$ using hashing:

  Create a hash index in s on the join attributes θ. Iterate thru r and for each distinct value of the join attribute, perfrom a hash lookup in s. If the hash lookup can't find an entry and return a `null` value, add the current r tuple to the result. And for edges cases like mentioned in $Semijoin$, the same general ideas apply.

12.15

Let the right relation be tr and the left relation be tl.

- Natural right outer join:

  For the natural right outer join. A similar strategy is applied:

  For the probe relation tr, if no matching tuple is found in the hash partition of it, it is treated with `null`s and included in the result.

  This can also be achieved by keeping a `boolean` flag with each tuple in the build relation `ret` and whenever any tuple in the probe relation tr matches with it, set the flag. When the probing is finished, all tuples in `ret` without a flag are marked as `null` and included in result.

- Natural full outer join:

  Do the above two operations together.

13.15

The best strategy is to:

Use indexing and locate the first tuple which `dept_name = 'music'`. Then retrieve the successive tuples using pointers as long as building is less than `Waston`. From the resulting tulles, reject those which do not satisfy `budge<55000`.

13.19

Suppose the histogram H storing distribution of values in r is divided into ranges $r_1, r_2, r_3, \ldots\ldots\ldots r_n$. For each range r_k with lowest value $r_{k:low}$ and highest value $r_{k:high}$, if $r_{k:high}$ is less than v, add the number of tuples with in $H(r_k)$ to the total. If $v < r_{k:high}\ and\ v > r_{k:low}$, assume values within r_k are uniformly distributed as no additional info is provided, add $H(r_k) \cdot \frac{v - r_{k:low}}{r_{k:high} - r_{k:low}}$ to the total.

13.24

a) Just sort r and retrieve the top K tuples, because these tuples are guaranteed to be contained in the result relation because join is on a foreign key of r referring s as they are unique.

b) Use a standard join on $r \bowtie s$ until the first K results are calculated. After K tuples in the result set, continue doing the standard join but discard any tuples from r that less than every tuples in the result set. If a newly joined tuple n has a attribute greater than any tuple in the result set, replace the lowest value tuple in result with t. It is going to be very inefficient cause a full join have to be executed.

13.25

Any query that only involves attribute A in r can be executed with index only.

An easy example with be:

```
select count(*)
from r
where A>500;
```