CS 245 – Winter 2002

Midterm solutions

Problem 1.

a.

Answer for Q1: **1 block**
Because we know values for both A and B, we can produce a single hash value for the query. This gives us one bucket to look in.

Answer for Q2: **$2^{10\text{-}X}$ blocks**
Because we only know values for A, we must try all possible values for the B portion of the hash value. Since the B portion is 10-X bits, there are $2^{10\text{-}X}$ possible values for the B portion of the hash. Thus, we need to examine $2^{10\text{-}X}$ blocks.

Common errors: Answering 2^X for Q1 or Q2.

b. **$0.4 + 0.6 \times 2^{10\text{-}X}$**
We multiply the cost for each query times the probability of that query.

Common errors: Usually people who got part a. right got this part right too.

c. **X=10**
The answer in part b is a function that increases as 10-X increases. In other words, the larger X is, the smaller the function in part b. Thus, with X at its maximum (10) the function is minimized.

Common errors: Not recognizing that the function in part b monotonically decreases with increasing X. Lots of people tried equating the derivative to zero and got stuck.

Problem 2.

a. **False**. You can generate multiple physical plans from a single logical plan by using different physical operators, join orders, etc.
b. **False**. Pushing down selections *sometimes* improves query processing, but only by reducing the sizes of intermediate results. The final query result should always be the same.
c. **True**.
d. **True**.
e. **False**. A join can be performed with a variety of non-index algorithms, for example, nested loop join.

f. **True**.
g. **True**.
h. **False**. In RAID-4, there is one redundant disk.
i. **False**. In a fixed format record, the schema tells us which field is which.
j. **True**.
k. **False**. For example, the rule that a disjunctive selection can be decomposed into the union of two selections is only true for setc.
l. **True**.
m. **False**. If one relation fits in memory, you can do a one pass join.

Problem 3.

a. **102,400 tracks.**

The maximum number of tracks is determined by the maximum allowable seek time. Setting $1+(n/2^9) = 201$ gives us n=102,400.

b. **8 surfaces.**
The number of tracks per surface (from part a) multiplied by the number of bytes per track (given in the problem) gives us the number of bytes per surface. The total number of bytes in the disk divided by the bytes per surface gives us 8 surfaces.

c. **10,000 RPM**

The maximum rotational latency is 6 milliseconds, which is thus the maximum rotation period. 1/6 ms, converted to minutes, gives us 10,000 RPM.

Common errors: Confusing rotational latency and seek time, calculation errors, assuming 6 milliseconds was the average rotational latency when in fact it is the maximum latency.

Problem 4.

a. **Depth = 3.**

Worst case depth happens when all (non-root) nodes are half full. Every leaf must have at least $\lfloor(100+1)/2\rfloor = 50$ keys. For a table with 20,000 records, this means we have 20,000/50 = 400 leaf nodes in the worst case. Every non-leaf, non-root node must have at least $\lceil 101/2\rceil = 51$ pointers. This means that in the worse case we have $\lceil 400/51\rceil = 8$ nodes in the layer above the leaf. The next layer up has 8 pointers and must be the root layer. The total is three layers.

Common errors: Working top down, which usually gave an answer of 4 layers.

b. **Depth = 2.**

In this case, the leaf nodes must have at least $\lfloor(200+1)/2\rfloor = 100$ keys. With 20,000 records we have 20,000/100 = 200 leaf nodes. Every non-leaf, non-root node must have at least $\lceil 201/2\rceil = 101$ pointers. This means that the layer above the leaf layer cannot have more than one node, or at least one node would have less than 101 pointers. Thus, the layer above the leaf layer is the root. The total is two layers.

Common errors: Again, working down, which gave an answer of 3 layers. Also, failing to realize that the layer above the leaf must be the root, and answering 3 layers.

c. **40 operations per second.**

If we use memory intelligently, we cache the root node of the B+-tree in one memory block and use the other as scratch storage. Since the B+-tree has 3 layers, we need to examine 3 B+-tree blocks and one data block for every operation. However, the root should already be in memory, so every operation requires only 3 I/Os. 120 I/Os per second / 3 I/Os per operation = 40 operations per second.

Also acceptable: arguing that in the worse case, we need to spend one I/O reading the root so we can cache it. Thus, the first operation requires 4 I/Os and every subsequent operation requires 3 I/Os. This gives an answer of **39 operations per second** for the first second, when we have to read the root block.

Common errors: not caching the root block, using the two buffers to do two operations in “parallel” as if this increased the number of I/Os per second to 240, answering “3 I/Os per operation” when the question asked for operations per second.

d. **60 operations per second.**

This answer is by the same argument in part c, except that the B+-tree now only has 2 layers, so only 2 I/Os per operation are needed. Also acceptable is to assume that the first operation must read the root block; this gives an answer of **59 operations per second.**

Common errors: usually if errors were made, they were the same errors in part c and d.

Problem 5.

a. **1,000 tuples**.

Since the natural join is over two attributes, we use the formula:

T(R)T(S)/(max(V(R,A),V(S,A))max(V(R,B),V(S,B))) =
2,000x200/(max(10,20)max(20,15)) =
400,000/(20x20) =
1,000

Common errors: most people got this right.

b. **2,222,222 tuples.**

The chance that a particular R tuple will join with a particular S tuple is the chance that the R.B set is a subset of the S.D set. This happens with probability

P = (Number of 2-element subsets of a given 5-element set)/(Total number of 2-element sets)

For a given value of S.D (a set with 5 elements), there are (5 choose 2) = 10 possible 2-element subsets. The attribute R.B can have as a value any one of (10 choose 2) = 45 possible sets of 2 elements. Thus, P = 10/45 = 2/9; each R tuple has a 2/9 chance of joining with any given S tuple. There are 1,000 S tuples, so every R tuple can expect to join with (2/9)(1,000) = 2,000/9 S tuples. There are 10,000 R tuples, so we can expect that the join will produce (2,000/9)(10,000) = 20,000,000/9 = 2,222,222.2222222 = 2,222,222 tuples.

Common errors: the probability in this problem is complicated, and many people had trouble with it.

Problem 6.

a. **x(4r + 2a)**

The worst case occurs when we compact x blocks and end up creating x overflow blocks. For each block that we compact we must read the block (cost: r), read the bucket we want to put those tuples into (cost: r), realize that we must allocate the overflow block, allocate the overflow block (cost: a), write the overflow block (cost: r), write the original bucket with a pointer to the overflow block (cost: r), and deallocate the block we are compacting (cost: a). The total is (4r + 2a) per compacted block, and there are x compacted blocks. Note: this analysis holds even if we create overflow chains of multiple overflow blocks. For example, imagine that we start with x+1 blocks and compact x blocks. In the worst case, each block we compact is full, so we cannot reuse an existing overflow block; we must create a new one. Thus, if we are clever, we can simply add the new block to the head of the overflow chain. The result is a linked list of x+1 blocks, and the cost is still x(4r+2a).

Also acceptable: we can optimize our algorithm by using the block we are about to deallocate as the new overflow block. This avoids one deallocation and one allocation, reducing the cost by 2a per block. The resulting cost is **x(4r)**.

b. **(y/2)(4r + 2a)**

Even though this is an extensible hash table, the cost per block is the same. Since the directory is always in memory, we simply have to perform the same sequence of operations to compact a block. This time, however, we compact y/2 blocks. Also acceptable is to use the optimization described in part a, to get **(y/2)(4r)** cost.

Common errors: Forgetting to take into account the cost to update pointers to the overflow blocks, assuming you can update a pointer to a block without reading the block first, not identifying the worst case correctly.