Database System Implementation

Solutions for Chapter 5

Solutions for Section 5.1

Exercise 5.1.1(a)

First, notice that if a rectangle does not have its upper-right corner in the northeast quadrant of the point (10,20), then it surely does not intersect the given rectangle. Likewise, if a rectangle does not have its lower-left corner in the southwest quadrant of the point (40,30), it cannot intersect. However, all rectangles that meet both conditions *do* intersect the given rectangle. An SQL query to check these conditions is:

```
SELECT id
FROM Rectangles
WHERE xur >= 10.0 AND yur >= 20.0 AND
      xll <= 40.0 AND yll <= 30.0;
```

Exercise 5.1.2(a)

```
SELECT color, COUNT(*)
FROM Sales
WHERE item = 'shirt'
GROUP BY color
HAVING COUT(*) > 1000;
```

Exercise 5.1.4

When we use the index for *x*, we retrieve 1000*n* pointers. These pointers are on about 5*n* leaf blocks. As in Example 5.5, we shall add one disk I/O for an intermediate block, and none for a leaf block. The index for YY requires the same number of B-tree block accesses, for a total of 10*n*+2 index blocks.

Because the data file is assumed sorted by *x*, the pointers in the intersection, of which there are *n*^2/1,000,000, are spread over fraction *n*/1000 of the blocks of the data file or 10*n* blocks. Thus, the total number of block accesses is 20*n*+2.

This number can be mor or less than reading all 10,000 data blocks. It is less, and therefore offers an advantage over a scan of the entire data file, if $n < 500$.

Exercise 5.1.5(a)

It is easier to calculate the probability that we will not be successful; it is the probability that each of 1,000,000 points chosen randomly and independently is outside a circle of radius *d*. The probability that any one point is outside the circle is 1 - pi*d*^2/1000000, so the probability that all 1,000,000 points are outside is that raised to the millionth power.

There is a useful approximation for small *x*, that *(1-x)^{a/x} = e^{-a}*. Thus, the approximate probability of failure is *e^{-pi*d^2}*, and the probability of success, i.e., that we shall find a point within radius *d* is *1 - e^{-pi*d^2}*.

Return to Top

Solutions for Section 5.2

Exercise 5.2.1(a)

Among the many ways to accomplish this task is to put one grid line in the hard-disk dimension at 8, and four grid lines in the speed dimension, at 280, 320, 360, and 420.

Exercise 5.2.4(a)

The best choice is to split according to speed, by introducing a grid line at 260 or thereabouts. That line divides two of the rectangles nontrivially, while any grid line in the RAM dimension will only split the one overflowing bucket.

Exercise 5.2.5(b)

The distance to the closest point found so far is a little more than 15. Points closer than that to the target point (110,205) can be found in five neighboring rectangles, those with lower-left corners at (80,200), (80,150), (100,150), (120,150), and (120,200).

Exercise 5.2.6(a)

It is possible to have 7 nonempty buckets, but no more. To see that there can be no more, notice that each of the 6 grid lines can only intersect the line of the data points once, and each intersection increases the number of line segments of the data points by 1. By choosing the grid points so that no two cross *on* the line of data points, we can in fact divide the data into 7 nonempty buckets.

Exercise 5.2.9

The approximate number of blocks/bucket is ceiling(*m*^2/100), because *m*^2 is the expected number of points in a square of side *m*, and there are 100 points per block. Since we assume the query boundaries do not coincide with a grid line, the expected number of stripes covered by the query in each dimension is 50/*m* + 1. Thus, the expected number of buckets that must be examined per query is (50/*m* + 1)^2.

Interestingly if we ignore the ceiling function, then the product of buckets per query and blocks per bucket is about 25, independent of *m*. However, because of the ceiling, we can never do this well. First, it doesn't make sense to pick *m* below 10, because the ceiling function tells us that we shall still require one block per bucket, no matter how small we make *m*.

At *m* = 10, the product is one block per bucket times 36 buckets/query = 36. We cannot do better than 36 by increasing *m*. To see why, remove the ceiling function to get a lower bound on the

number of blocks per query: *(m^2/100)(50/m + 1)^2 = 25 + m + m^2/100*. Since *m* >= 10, the above expression is at least 36.

Thus, we conclude that *m* = 10, and 36 blocks per query is the best we can do.

Return to Top

Solutions for Section 5.3

Exercise 5.3.3(a)

For the index on *x* we need 10 first-level index blocks to hold all 100 values of *x* and their corresponding pointers to indexes on *y*. Thus, we need a second-level index with one block. Finding the *y*-index for the *x*-value *x=C* thus requires two disk I/O's.

Now, we must enter the *y*-index for this *x*-value. Since there are 10 *y*-values for each *x*-value, this index is a single block, which must be retrieved. We have now made three disk I/O's. If there is indeed a record with *x=C* and *y=D*, then a fourth disk I/O is needed to retrieve the *z*-value.

Exercise 5.3.5(a)

To get to the block with (30,260), you have to go right at a node with salary = 150 and left at a node with salary = 300. Thus the salary must be at least 150, and less than 300. Also, you must go left at a node with age = 47, so the age must be less than 47.

Exercise 5.3.10

Nope; can't do it. Consider the case where all the points are in a line. We can split that line into three parts by the x- and y-axes, but we can't split it into four parts. Thus, one quadrant is always empty.

Return to Top

Solutions for Section 5.4

Exercise 5.4.1(a)

First, the computers A through L will correspond to positions 1 through 12 of the bit vectors, respectively. Then the bit vectors for speed are:

233: 000000010000
266: 000000001000
300: 100000000100
333: 010000000000
350: 000100000010
400: 001001000001
450: 000010100000

Exercise 5.4.5(a)

First, let's translate the bit vector into ``runs." The lengths of the runs of 0's preceding the 1's are 1, 0, 7, and 5. The codes for these numbers are 01, 00, 110111, and 110101, respectively, so the encoding is 0100110111110101.

Exercise 5.4.6

Suppose that pointers require p bits, and keys require k bits. Since each of the n records has a key and pointer in the B-tree, the leaves alone occupy at least $n(k+p)$ bits. If there are, say, 100 key-pointer pairs in a block, then the total space of the B-tree is only about 1% more than the space of the leaves, so we can neglect all but the space of the leaves.

We must thus compare 2log_2n with $k+p$, since these are the multipliers of n in the formulas for space taken by bitmap indexes and B-trees, respectively. The contribution of k is a little tricky. If the key has many bits, but there are few different values of the key in the file (a common situation), then the space for listing the different key values in the bitmap index, which we ignored in Section 5.4.2, is much less than the space for representing keys in the B-tree. However, if the number of actual key-values in the file approximates the 2^k, then the term nk in the formula for B-tree space can be deleted, since to be fair we should have added the same term into the cost of the bitmap index.

Let us, to be conservative, compare only p with 2log_2n. If there are n records in the file, surely pointers must take at least log_2n bits. However, in practice, pointers must do more than distinguish among records of one file; they must distinguish locations on many disks, and therefore 32 bits is a minimum for pointers, with considerably higher values possible. If $p = 32$, then the bitmap index will take less space if 2log_2n < 32, or log_2n < 16, or n < 65,536.

Thus, for a reasonable sized file, the space required by the bitmap and B-tree are comparable under our assumptions above. The advantage goes to the bitmap index in a number of situations, including:

1. Pointers larger than 32 bits.

2. Keys require many bits, but the number of different key values is small.

There are a number of advantages to the B-tree, even when it takes more space, including the ability to handle range queries efficiently and the speed with which access to single records is provided.

Return to Top