Storage and Indexing

1. To enforce the constraint that eid is a key, all we need to do is make the clustered index on eid unique and dense. That is, there is at least one data entry for each eid value that appears in an Emp record (because the index is dense). Further, there should be exactly one data entry for each such eid value (because the index is unique), and this can be enforced on inserts and updates.
2. If we want to change the salaries of employees whose eid's are in a particular range, it would be sped up by the index on eid. Since we could access the records that we want much quicker and we wouldn't have to change any of the indexes.
3. If we were to add 1 to the ages of all employees then we would be slowed down, since we would have to update the index on age.
4. 4. If we were to change the sal of those employees with a particular did then no advantage would result from the given indexes.

Disks and Files

1. 1024/100 = 10. We can have at most 10 records in a block.
2. There are 100,000 records all together, and each block holds 10 records. Thus, we need 10,000 blocks to store the file. One track has 25 blocks, one cylinder has 250 blocks. we need 10,000 blocks to store this file. So we will use more than one cylinders, that is, need 10 surfaces to store this file.
3. The capacity of the disk is 500,000K, which has 500,000 blocks. Each block has 10 records. Therefore, the disk can store no more than 5,000,000 records.
4. There are 25K bytes, or we can say, 25 blocks in each track. It is block 26 on block 1 of track 1 on the next disk surface.
If the disk were capable of reading/writing from all heads in parallel, we can put the first 10 pages on the block 1 of track 1 of all 10 surfaces. Therefore, it is block 2 on block 1 of track 1 on the next disk surface.
5. A file containing 100,000 records of 100 bytes needs 40 cylinders or 400 tracks in this disk. The transfer time of one track of data is 0.011 seconds. Then it takes 400 × 0.011 = 4.4seconds to transfer 400 tracks.
This access seeks the track 40 times. The seek time is 40 × 0.01 = 0.4seconds.
Therefore, total access time is 4.4+ 0.4 = 4.8 seconds.
If the disk were capable of reading/writing from all heads in parallel, the disk can read 10 tracks at a time. The transfer time is 10 times less, which is 0.44 seconds.
Thus total access time is 0.44 + 0.4 = 0.84 seconds
6. For any block of data, averageaccesstime = seektime+rotationaldelay+transfertime.
seektime = 10msec
rotationaldelay = 6msec
transfertime = 1K/(2, 250K/sec)= 0.44msec
The average access time for a block of data would be 16.44 msec. For a file containing 100,000 records of 100 bytes, the total access time would be 164.4 seconds.

Tree-Structured Indexing

1. Since the index is a primary dense index, there are as many data entries in the B+ tree as records in the heap file. An index page consists of at most 2d keys and 2d+1 pointers. So we have to maximize d under the condition that $2d \cdot 40+(2d+1) \cdot 10 \leq 1000$. The solution is d = 9, which means that we can have 18 keys and 19 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 40+10=50 bytes. Therefore a leaf page has space for (1000/50)=20 data entries. The resulting tree has $\log_{19}(20000/20)+ 1 = 4$ levels.
2. Since the nodes at each level are filled as much as possible, there are 20000/20 = 1000 leaf nodes (on level 4). (A full index node has 2d+1 = 19 children.) Therefore there are 1000/19 = 53 index pages on level 3, 53/19 = 3 index pages on level 2, and there is one index page on level 1 (the root of the tree).
3. Here the solution is similar to part 1, except the key is of size 10 instead of size 40. An index page consists of at most 2d keys and 2d+1 pointers. So we have to maximize d under the condition that $2d \cdot 10 + (2d + 1) \cdot 10 \leq 1000$. The solution is d = 24, which means that we can have 48 keys and 49 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 10+10=20 bytes. Therefore a leaf page has space for (1000/20)=50 data entries. The resulting tree has $\log_{49}(20000/50)+ 1 = 3$ levels.
4. Since each page should be filled only 70 percent, this means that the usable size of a page is $1000 \cdot 0.70 = 700$ bytes. Now the calculation is the same as in part 1 but using pages of size 700 instead of size 1000. An index page consists of at most 2d keys and 2d + 1 pointers. So we have to maximize d under the condition that $2d \cdot 40+(2d+1) \cdot 10 \leq 700$. The solution is d = 6, which means that we can have 12 keys and 13 pointers on an index page. A record on a leaf page consists of the key field and a pointer. Its size is 40+10=50 bytes. Therefore a leaf page has space for (700/50)=14 data entries. The resulting tree has $\log_{13}(20000/14) + 1 = 4$ levels.

Hash-Based Indexing

1. If we start with an index which has B buckets, during the round all the buckets will be split in order, one after the other. A hash function is expected to distribute the search key values uniformly in all the buckets. This kind of split during the round causes a redistribution of key values in all the buckets. If a bucket has overflow pages, after the redistribution it is likely that the length of the overflow chain reduces. If the hash function is good, the length of the overflow chains in most buckets is zero because in each round there will be at least one redistribution of the values in each bucket. The number of overflow pages during the round is not expected to go beyond one because the hash function distributes the incoming entries uniformly.
2. No. Overflow chains are part of the structure, so no such guarantees are provided.
3.

N/0.8P
This can be achieved when all the keys map into the same bucket. (The effect of 80% occupancy is to increase the number of pages in the file, relative to a file with 100% occupancy.)
4. Consider the index in Fig 11.6. Let us consider a list of data entries with search key values of the form 2i where i > k. By an appropriate choice of k, we can get all these elements mapped into Bucket0. Suppose we have m primary data pages, each time we need to add one more overflow page to Bucket0, it will cause a page split. So if we add n overflow pages to Bucket0, the space utilization = (n + 1)/(m+ n + n), which is less than 50%.