12.10 Suppose you need to sort a relation of 40 gigabytes, with 4 kilobyte blocks, using a memory size of 40 megabytes. Suppose the cost of a seek is 5 milliseconds, while the disk transfer rate is 40 megabytes per second.

a. Find the cost of sorting the relation, in seconds, with bb = 1 and with bb = 100.

Ans: b_r = 40 GB / 4 KB = 10000000 blocks

M = 40 M B / 4 K B = 10000 blocks

The initial number of runs = (b_r / M) = 1000

b. In each case, how many merge passes are required?

Ans: The number of merge passes required = ($\log_{M-1}$ (b_r / M))= ($\log_{9999}$ 1000) = 1

Block transfers = b_r (2 ∗ 1 + 1) = 30000000 block

Seeks = 2 (b_r / M) + (b_r / bb) (2 ∗ 1 − 1)

Therefore, if bb= 12000 + 10000000 = 10002000 seeks

bb= 1002000 + 100000 = 102000 seeks

Total sorting cost in seconds = (hash(#) of block transfers) * 4KB / 40MB + (hash(#) of seeks) * 5/1000

c. Suppose a flash storage device is used instead of a disk, and it has a seek time of 1 microsecond, and a transfer rate of 40 megabytes per second. Recompute the cost of sorting the relation, in seconds, with bb = 1 and with bb = 100, in this setting.

Ans: if bb= 130000000 ∗ 4 KB / 40 MB + 10002000 ∗ 5 / 1000 = 3000 + 50010 = 53010 sec

bb= 10030000000 ∗ 4 KB / 40 MB + 102000 ∗ 5 / 1000 = 3000 + 510 = 3510 sec

12.12 Why is it not desirable to force users to make an explicit choice of a query processing strategy? Are there cases in which it is desirable for users to be aware of the costs of competing query-processing strategies? Explain your answer.

Ans: In general it is not desirable to force users to choose a query processing strategy because naive users might select an inefficient strategy. The reason users would make poor choices about processing queries is that they would not know how a relation is stored, nor about its indices. It is unreasonable to force users to be aware of these details since ease of use is a major object of database query languages. If users are aware of the costs of different strategies, they could write queries efficiently, thus helping performance. This could happen if experts were using the system.

12.13 Design a variant of the hybrid merge-join algorithm for the case where both relations are not physically sorted, but both have a sorted secondary index on the join attributes.

Ans: We merge the leaf entries of the first sorted secondary index with the leaf entries of the second sorted secondary index. The result file contains pairs of addresses, the first address in each pair pointing to a tuple in the first relation, and the second address pointing to a tuple in the second relation.

This result file is first sorted on the first relation's addresses. The relation is then scanned in physical storage order and addresses in the result file are replaced by the actual tuple values. Then the result file is sorted on the second relation's addresses, allowing a scan of the second relation in physical storage order to complete the join

12.14 Estimate the number of blocks transfers and seeks required by your solution to Exercise 12.13 for r1 $\bowtie$ r2, where r1 and r2 are as defined in Practice Exercise 12.3.
Ans: r1 occupies 800 blocks, and r2 occupies 1500 blocks. Let there be n pointers per index leaf block (we assume that both the indices have leaf blocks and pointers of equal sizes). Let us assume M pages of memory, M < 800. r1's index will need B1 = (20000/n) leaf blocks, and r2's index will need B2 = (45000/n) leaf blocks. Therefore, the merge join will need B3 = B1 + B2 accesses, without output. The number of output tuples is estimated as no = (20000∗45000 /max (V (C,r1),V (C,r2))). Each output tuple will need two pointers, so the number of blocks of join output will be Bo1 = (no n/2). Hence the join needs Bj = B3+Bo1 disk block accesses. Now we must replace the pointers by actual tuples. For the first sorting, Bs1 = Bo1(2 (logM−1(Bo1/M)) + 2) disk accesses are needed, including the writing of output to disk. The number of blocks of r1 which must be accessed in order to replace the pointers with tuple values is min (800, no). Let n1 pairs of the form (r1 tuple, pointer to r2) fit in one disk block. Therefore, the intermediate result after replacing the r1 pointers will occupy Bo2 = (no/n1) Blocks.
Hence the first pass of replacing the r1-pointers will cost Bf = Bs1 + Bo1 + min (800, no) + Bo2 disk accesses. The second pass for replacing the r2-pointers has a similar analysis. Let n2 tuples of the final join fit in one block. Then the second pass of replacing the r2-pointers will cost Bs = Bs2 + Bo2 + min (1500, no) disk accesses, where Bs2 = Bo2(2 (logM−1(Bo2/M)) + 2). Hence the total number of disk accesses for the join is Bj + Bf + Bs, and the number of pages of output is (no/n2).

12.15 The hash-join algorithm as described in Section 12.5.5 computes the natural join of two relations. Describe how to extend the hash-join algorithm to compute the natural left outer join, the natural right outer join and the natural full outer join. (Hint: Keep extra information with each tuple in the hash index, to detect whether any tuple in the probe relation matches the tuple in the hash index.) Try out your algorithm on the takes and student relations.
Ans: For the probe relation tuple tr under consideration, if no matching tuple is found in the build relation's hash partition, it is padded with nulls and included in the result. This will give us the natural left outer join tr ⟕ ts. To get the natural right outer join tr ⟖ ts, we can keep a Boolean flag with each tuple in the current build relation partition H_{si} residing in memory and set it whenever any probe relation tuple matches with it. When we are finished with H_{si}, all the tuples in it which do not have their flag set, are padded with nulls and included in the result. To get the natural full outer join, we do both the above operations together. To try out our algorithm, we use the sample customer and depositor relations of Figures 13.17 and 13.18. Let us assume that there is enough memory to hold three tuples of the build relation plus a hash index for those three tuples. We use depositor as the build relation. We use the simple hashing function which returns the first letter of customer-name. Taking the first partitions, we get Hr1 = {("Adams", "Spring", "Pittsfield")}, and Hs1 = ϕ. The tuple in the probe relation partition will have no matching tuple, so ("Adams", "Spring", "Pittsfield", null) is outputted. In the partition for "D", the lone build relation tuple is unmatched, thus giving an output tuple ("David", null, null, A-306). In the partition for "H", we find a match for the first time, producing the output tuple (”Hayes”, ”Main”, ”Harrison”, A-102). Proceeding in a similar way, we process all the partitions and complete the join.

12.18 Suppose you have to compute AGsum(C)(r) aswell as A,BGsum(C)(r). Describe how to compute these together using a single sorting of r.

Ans: The estimated size of the relation can be determined by calculating the average number of tuples which would be joined with each tuple of the second relation. In this case, for each tuple in r1, 1500/V (C, r2) = 15/11 tuples (on the average) of r2 would join with it. The intermediate relation would have 15000/11 tuples. This relation is joined with r3 to yield a result of approximately 10,227 tuples (15000/11 × 750/100 = 10227). A good strategy should join r1 and r2 first, since the intermediate relation is about the same size as r1 or r2. Then r3 is joined to this result.