Solver: Jesslyn Chew

1)

a)

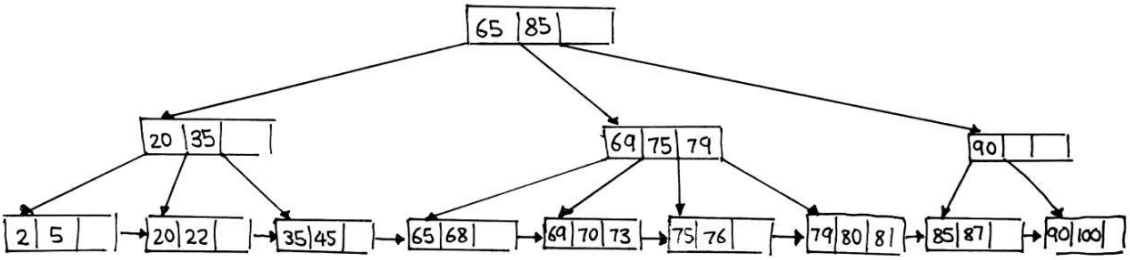
i) The 3 components are seek time, rotational delay and transfer time. The transfer time takes the least time for a random I/O.

ii) I/O Cost = 1000 ÷ 300
= 3.33
≈ **4** (round up)

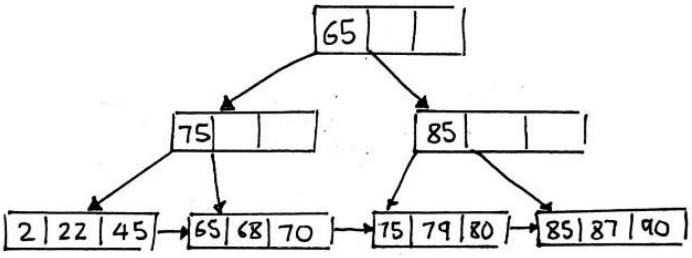
iii) Double buffering algorithm enables the buffer pool to process one buffer in the memory while doing an I/O read for another buffer to prepare for the next processing step.

iv) The 2 options are physically contiguous order or using linked list. The first option means that the blocks are stored in a consecutive order in the data block. Whereas, the second option enables the block to be sequenced logically where a linked list is used to indicate the next set of records without the head to physically move data to keep the sequential ordering when the database is modified.

b)



i)



ii)

iii) At the 1st level, key = 65 > 39, hence it will search the left subtree. On the 2nd level, the first key = 20 < 39, so the first subtree (most left) is ignored. The next key = 35 < 39, so the second subtree will be ignored as well. Hence, it will search the last subtree of that node. From the leaf node, there are 2 keys and 2 pointers. The first key = 35 > 39 and the second key = 45 > 39. Since it is sparse index, it will explore the first pointer. Total I/O Cost = 3 + 1 = **4**, where the "1" is the cost for accessing the actual record.

iv) It will look for the ley value = 39 as explained in part iii. It will access the leaf pointers of all the keys on the right side of the "35" key. If it is the end of the node, it will access the incremental pointer and access the adjacent leaf node on its right. It will stop when the key value is more than 76 or it has reached the end of the leaf nodes.

v) Let n be the # of levels and k be # of keys per node
Best case: the key value in the range of [39, 76] does not exist
I/O Cost of best case = n

Worst case: all keys in the B+ tree is in range of [39,76] and the tree is full.
I/O Cost of worst case = # of levels + number of points in leaf nodes
= $n + k(k + 1)^{n-1}$

2)

a)

i) Static hashing has a fixed number of buckets while extensible hash index can have a varied number of buckets. Extensible hash index bucket size is fixed to 1 block per bucket while static hash index bucket size can vary. For extensible hash index, each bucket may have more than 1 hash value assigned to it while static hash index only has 1 hash value to each bucket.

ii) There can be a maximum of 16 buckets, where each bucket only has 1 hash value assigned to it. Based on the hash function, there can be 16 different values that can be formed from h(n).

iii) Sparse index implies that each leaf node pointer points to 1 block.
# of leaf node pointers = $101^2 * 100$
= 1,020,100
# of tuples = 1,020,100 * 20
= **20,402,000** tuples

b)

i) Indexes can help to reduce the time cost for searching a key with a specific value.

```
SELECT *
FROM people as p
WHERE p.id = 5;
```

Indexes can also help to reduce the time cost for searching records with an attribute that lies within a specified range.

```
SELECT *
FROM nation as n
WHERE n.population < 10000 AND n.population > 100;
```

ii) The first disadvantage is that updating and insertion will be slower to keep index in order. The second disadvantage is that more space is required to store the index file.

c) This clause enables us to ensure some space of the nodes in the B+ tree are empty, making room for future insertions and updates. Hence, reducing the time for insertions and updates by reducing the need to reorder the tree that occurs when there is insufficient space in the nodes that the key is supposed to be inserted or updated to.

d)

i) Assume there are M buffers in the memory. M-1 buffers are used to fit the smaller relation. The smaller relation must be a size of M-1 blocks or smaller. The M^{th} buffer is used to iterate the blocks in the larger relation. For each block in the larger table, each tuple is compared with all the tuples of the smaller relation in memory. If the join attribute(s) are the same in both tuples, the tuples will be joined and stored in the output buffer.

ii) Block-based Nested loop algorithm can be used to join R2 and R3 together. Any of the relation can be the outer loop since the sizes are the same. The intermediate results are written back into the disk. Since B(R1) = 10 < M, we can use one-pass join algorithm with R1 and the intermediate results of the previous join. The one-pass join algorithm is explained in

part [i].

I/O Costs for block-based nested loop $= \left\lceil \frac{B(R2)}{M-1} \right\rceil * B(R3) + B(R2)$
$= \left\lceil \frac{100}{20} \right\rceil * 100 + 100$
$= 600$

We assume the worst case, where the intermediate result size is B(R2)*B(R3).

I/O Costs for one-pass = B(R1) + B(intermediate result)
= 10 + 100*100
= 1010

Total I/O Costs = 1610

3)

a)

i) [Not too sure on the disk I/O cost for set union as the lecture slides didn't explicitly mentioned, but I assume it is block-based nested loop join as it seems the most suitable]

I/O Cost $= \left\lceil \frac{B(R)}{M-1} \right\rceil * B(S) + B(R)$
$= \left\lceil \frac{10000}{1000-1} \right\rceil * 10000 + 10000$
$= \mathbf{12000}$

ii) I/O Cost = 3(B(R) + B(S))
= 3(20000)
= **60,000**

b)

i) Space requirement: min(B(R), B(S)) ≤ M^2
min (10000, 20000) ≤ M^2
M ≥ $\sqrt{10000} = 100$
The minimum number of blocks for main memory buffer is **100**.
I/O Cost = 3(B(S) + B(R))
= 3(10000 + 20000)
= **90,000**

ii) The average number of blocks having a distinctive value of a is f(k) = B(R) ÷ k = 10000 ÷ k
Since we have to read blocks containing k/10 distinctive values of a, the number of blocks to read will be: (10000 ÷ k) (k ÷ 10) = 1000

c) <No solution provided>

d)

i) Pipelining means that each query operator makes its output tuples available to the next operator as soon as they are produced, rather than waiting until it has finished producing all of its output tuples.

ii) A blocking operator consumes all of its input tuples before producing any output tuples

4)

a)

i) Start checkpoint after logID 11. <START CKPT (T3, T4)>
End checkpoint after logID 15. <END CKPT>

ii) Need to inspect the transactions after the checkpoint, specifically the transactions that are not committed since it is using an UNDO log. After logID 16, T5 is not committed yet so it needs to be undone. From logID 16, the database need to set A = 6.

iii) Start checkpoint after logID 4. <START CKPT (T1, T2)>
End checkpoint after logID 6. <END CKPT>

iv) Redo the transactions that have been committed before the crash (logID 16). Hence, need to redo transaction T1, T2, T3 and T4. Need to inspect from the start of earliest transaction mentioned in the checkpoint, which is logID 1 (T1). Need to redo logID 2, 4, 6, 9, 11, 13.

v) Redo the transactions that have been committed before the crash (logID 6). Hence, need to redo transaction T1 only. Need to inspect from the start of earliest transaction mentioned in the checkpoint, which is logID 1 (T1). Need to redo logID 2 and 4.

b) The schedule is not serializable. As the is a lost update in B, consistency is not maintained in the database. This implies that the schedule is not serializable, as a schedule is serializable if it is equivalent to a serial schedule and a serial schedule ensure consistency.

Interleaved schedule		Serial schedule	
T1	T2	T1	T2
R(A)		R(A)	
	R(B)	W(A)	
	R(C)	R(B)	
W(A)		W(B)	
R(B)		Commit	
W(B)			R(B)
	W(C)		R(C)
	W(B)		W(C)
Commit			W(B)
	Commit		Commit

c)

i) The request is accepted – a

ii) The transaction is rolled back – d

iii) The request is ignored – b

iv) The request is accepted – a

--End of Answers--