1)

a)

i) Transfer time

ii) Query time (insert, update, delete)

iii) A because more records can fit into one block

iv) Yes, because it has one index entry for each data record

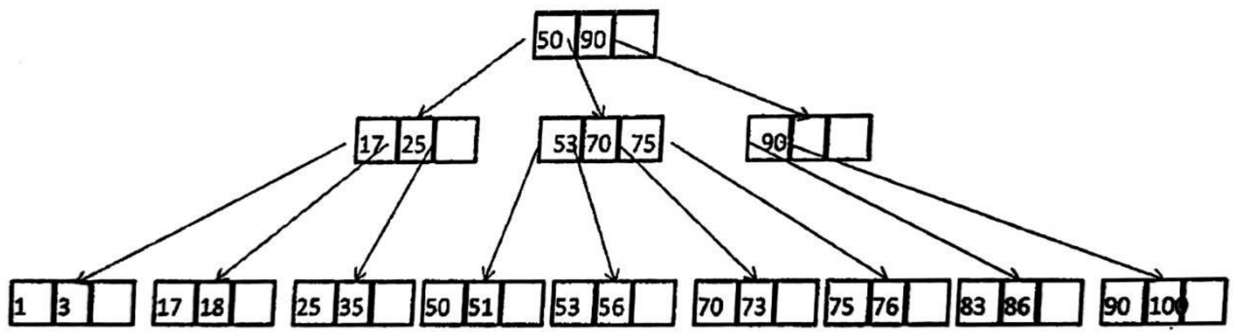
b) Since prefetching algorithm is used,

I/O cost = R + P + (# of blocks – 1) * max (R, P)

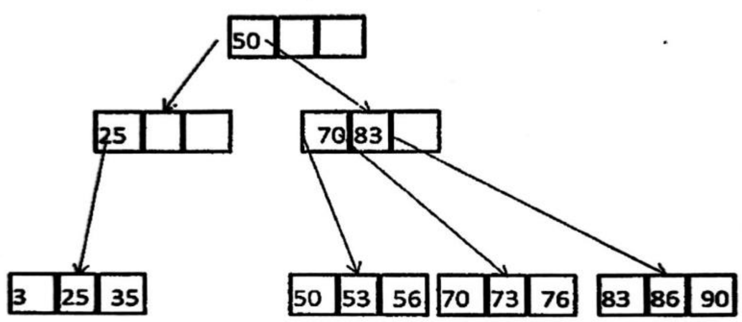
= **R + 4P** [since R < P]

c)

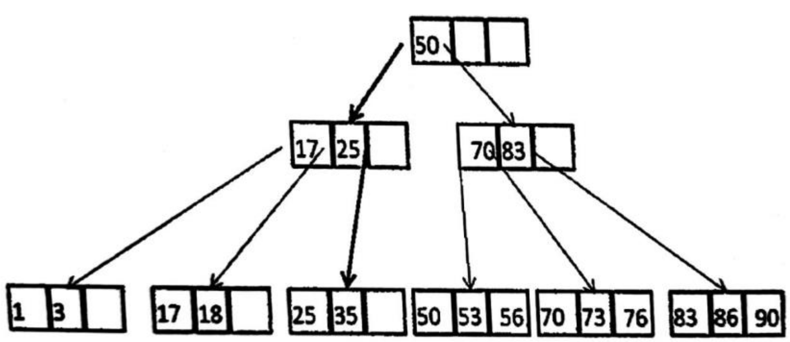
i)



ii)



iii)



The bold arrows indicates the search path.

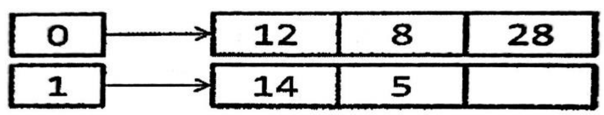
d) $(19+1)^{h-1}*19 \geq 1000$
$h \geq 3$
Hence, the minimum height of the B+ Tree is 3

e) For data, need 100000/10 = 10000 blocks
For index,
$79^{h-1}*80 \geq 100000$
h = 3 (h is the height of B+ tree)
Need $1+80+80^2$ = 6481 blocks
Totally need 16481 blocks

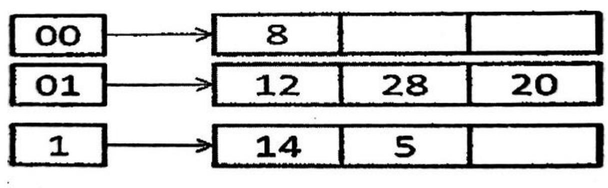
2)

a)

i)



ii)



b) 32*10*5 = 1600 tuples

c) Yes.
Require less index space.
Update operations (insert, delete) are easier and more efficient.

d) B+ tree index (tale advantage of range query) on the attribute ‘a’, clustered (take advantage of range query)
B+ tree index (take advantage of range query) on the attribute ‘c’, clustered (take advantage of range query)

e) B(R1) = 100000/200 = 500 blocks
B(R2) = 50000/500 = 100 blocks
M = 101
Using block-based nested loop join
minCost = B(R2) + B(R1) * upperBound(B(R2)/(M-1)) = 600 block access

3)

a)

i) B(R) = 1500, B(S) = 460, M = 101
$B(R) + B(S) < M^2$ → Refined sort-merge join algorithm can be applied
I/O cost = 3 * (B(R) + B(S)) = 5880 disk I/O

ii) Read 1500 block of R, sort, write back 150 sorted sublists → cost 1500*2 = 3000 I/O
Read 460 block of R, sort, write back 4 sorted soblists, the last sublist contains 60 last block will be kept in memory → cost 460 + 400 = 860 I/O
Merge phase: read blocks in sublists of R and S linearly → cost 1500 + 400 = 1900 I/O
Total cost 3000 + 860 + 1900 = 5760 I/O
Number of I/O(s) saved: 5880 – 5760 = 120 I/O
Actually the number of I/O saved is 2 * size of sublist kept in memory

iii) Cost of index-based join: B(S) + T(S)*T(R)/V(R,'x') = 460 + 9200 * 15000 / 50
Cost of hash join: 3(B(S) + B(R)) = 3 * (460 + 1500)
Hence index-based join is a much worse choice in the case.

iv) T(R) = 15000, T(S) = 9200
Number of tuples satisfy y > 5: T(R) / 3 = 5000 → X
Number of tuples satisfy z ≥ 9: 0.2 * T(S) = 1840 → Y
(0.2 since 1 ≤ z ≤ 10 and z values are evenly distributed in S)
Join of X and Y: T(X) * T(Y) / max (V(X,'x'), V(Y,'x')) = 5000 * 1840 / 50 = 184,000

b) **Materialization**
Order the operations (an operation is not performed until the argument(s) below have been performed)
Store the result of each operation on disk until it is needed by another operation
Pipelining
Interleave the execution of several operations
The tuples produced by one operation re passed directly to the operation that it
No need to store intermediate relation in disk
Several operations must share main memory at any time

4)

a) **Atomicity**: A transaction is either performed in its entirety (all its steps) or not performed at all
Correctness: If T starts with DB in consistent state then T executes in isolation, T leaves consistent state
Isolation: A transaction should appear as though it is executed in isolation from other transactions
Durability: Changes applied to the database by a committed transaction must persist in the database

b)

i) T1 does not need to be redone because it is committed in LogID4 and successfully written to disk after <END CKPT>

T2 needs to be redone because it's uncommitted (its commit log is LogID10 which is after the crash point)

ii) T1 is done hence (T1, A, 2) is updated on disk
If T2 successfully write to disk before the crash point → (A, B, C) = (3, 4, 6) (i.e. all changes by T1 and T2 are written to disk)
If T2 NOT successfully write to disk before the crash point → (A, B, C) = (2, 1, 1) (i.e. only changes by T1 and written to disk)

c) Deadlock: the situation when each set of transactions is waiting for a resource (e.g. lock) currently held by another transaction in the set. None can make progress. Example:
T1: $l_2(A)l_2(B)w_1(B)$ …
T2: $l_2(B)l_2(A)w_2(A)$ …
T1 holds lock on A and request for lock on B, while T2 holds lock on B and request for lock on A. None of them releases the needed lock → Keep waiting

d) **Undo logging**
Requires that data be written to disk immediately after a transaction finishes
Increase number of disk I/Os
Undo all uncommitted transactions, bottom-up (or latest-first)
Redo Logging
Requires to keep all modified blocks in buffers until the transaction commits and the log records have been flushed
Increases average number of buffers needed by transactions
Redo all committed transaction, top-down (or earliest-first)

--End of Answers--

Solver: Phan Cong Minh