1. (Solved and Graded by: Xiaojuan)
   Note: may accept other correct answers.

   a. "mining": Repeatedly compute a hash problem to find an answer below some threshold T. Once such an answer is found, a block is considered "mined" and the miner can become the leader for that block. "proof of work" (POW): A consensus protocol for validating transactions and generating new blocks. Miners need to solve a puzzle that can only be solved by brute force. Therefore, a valid block is proof of "hard" work.

   b. A blockchain is a distributed ledger. It uses a consensus algorithm to choose the next block of transactions to append to the history chain, so everyone agrees on the same transaction history (chain of blocks).

   c. A major difference is that everyone can join a permissionless blockchain and remain anonymous if preferred. There are trusted nodes in permissioned blockchains, and they are typically run by companies. This also means that cheaper consensus algorithms can be run among trusted nodes that don't require all nodes to participate.

   d. "proof of stake" uses less energy. In "proof of work", miners need to solve complex puzzles using brute force, which usually costs a large amount of computation. In "proof of stake", the protocol is based on the stake validators put into the system. Therefore no extra computation is needed.
2. (Solved and Graded by: Adit.)
First, we will list all the pairs of conflicting operations between T1 and T2:

1. read(a, T1) — write(a, baz, T2)
2. [2nd] read(a, T1) — write(a, baz, T2)
3. write(b, caz, T1) — read(b, T2)
4. write(b, caz, T1) — write(b, bar, T2)
5. write(c, foo, T1) — read(c, T2)

a. For this interleaving, these are the orders in which the above pairs are executed:

1. (T1, T2)  2. (T1, T2)  3. (T2, T1)  4. (T1, T2)  5. (T1, T2)

Since the 3rd pair is executed in a different order than all the other pairs, this interleaving is NOT serially equivalent.

b. Following the same logic:

1. (T1, T2)  2. (T2, T1)  3. (T2, T1)  4. (T2, T1)  5. (T2, T1)

Since the 1st pair is executed in a different order than all the other pairs, this interleaving is NOT serially equivalent.

c. 1. (T1, T2)  2. (T1, T2)  3. (T2, T1)  4. (T1, T2)  5. (T1, T2)

Since the 3rd pair is executed in a different order than all the other pairs, this interleaving is NOT serially equivalent.

d. 1. (T1, T2)  2. (T1, T2)  3. (T1, T2)  4. (T1, T2)  5. (T2, T1)

Since the 5th pair is executed in a different order than all the other pairs, this interleaving is NOT serially equivalent.

e. No, the interviewer is incorrect. Take the following counterexample:
read(a, T1); read(b, T2); write(b, bar, T2); write(a, baz, T2); read(a, T1); read(c, T2);

In this interleaving, T1’s first read on ‘a’ occurs before T2’s write on ‘a’, while T1’s second read on ‘a’ occurs afterwards. This is clearly a violation of serial equivalence, and therefore the interviewer’s statement is false.
3. (Solved and Graded by: Kshitij.)

We know that each object is represented by a unique word-id. If locks are acquired in a lexicographically decreasing order, we are correct in saying that deadlocks will not occur even if locks are acquired during the transaction.

We build a wait for graph for all transactions in the system and the assumption here is that all transactions obey our algorithm.

**Proof Using Contradiction:**

Let us assume that there is a Deadlock in the system. What this means is that there is a cycle present somewhere in the wait for graph. Now if the rules are obeyed by the transactions, every transaction T must be waiting to acquire a lock on an Object (O) with a lower-id than the objects it has already acquired locks on (Since Objects with higher-id are locked first as per the algorithm)

But Since there is a deadlock in the system, the transaction is waiting to acquire a lock on an object with a higher id which leads to the formation of cycle in the graph.

This is not possible since all of the transactions are supposed to obey the algorithm. Hence we have a contradiction and there is no cycle in the graph which also means that there is no deadlock in the system.

**Updated Proof:**

Consider N transactions in a cycle: T1, .. Tn. Suppose that all the transactions were following the described locking scheme which locks objects in a decreasing order. This means that every transaction Ti in the circular wait cycle should be waiting on an object with a lower id (i.e., (Oi-1) than the objects it has already acquired (Oi)). But because O1...On are in a cycle, this means that O1 has both a lower, and a higher ID than On, which is a contradiction. This shows that it is impossible for circular wait to happen (e.g., O1 < O2 < On < O1) and hence no deadlocks should occur.
4. (Solved and Graded by: Fangqi Han.)
   a. It no longer satisfies serial equivalence as allowing lock acquisition during or after
      the shrinking phase breaks the two-phase locking proof.

      Here is a counterexample:

      | Transaction T1 | Transaction T2 |
      |----------------|----------------|
      | Lock(O1)       |                |
      | Write(O1)      |                |
      | Unlock(O1)     | Lock(O1)       |
      | Write(O1)      | Unlock(O1)     |
      | Lock(O2)       | Lock(O2)       |
      | Write(O2)      | Write(O2)      |
      | Unlock(O2)     | Unlock(O2)     |
      | Lock(O2)       |                |
      | Write(O2)      |                |
      | Unlock(O2)     |                |

      T1 and T2 have conflicting operations. For object O1, the time ordering pair is
      (T1, T2), but for object O2 the pair is (T2, T1). This violates serial equivalence.

   b. It can still deadlock, same as the original two-phase locking.

      Example:

<pre><code>  | Transaction T1 | Transaction T2 |
  |----------------|----------------|
  | Lock(O1)       | Lock(O2)       |
</code></pre>
<table>
<thead>
<tr>
<th>Lock(O2)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lock(O1)</td>
</tr>
</tbody>
</table>

It enters a deadlock once T1 acquires lock for O1 and T2 acquires lock for O2. Both T1 and T2 will block and wait for the other indefinitely.
5. (Solved and Graded by: Ritwik Deshpande.)

In the above question the lock is acquired before the first access and relinquished after the last access of the object

a. Serial equivalence: Using counter-example we can show that it is not serially equivalent.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock1(A)</td>
<td></td>
</tr>
<tr>
<td>W1(A)</td>
<td></td>
</tr>
<tr>
<td>Rel1(A) -&gt; Last access of obj A</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock2(B)</td>
</tr>
<tr>
<td></td>
<td>W2(B)</td>
</tr>
<tr>
<td></td>
<td>Rel2(B) -&gt; Last access of obj B</td>
</tr>
<tr>
<td>Lock1(B)</td>
<td></td>
</tr>
<tr>
<td>R1(B)</td>
<td></td>
</tr>
<tr>
<td>Rel1(B) -&gt; Last access of obj B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock2(A)</td>
</tr>
<tr>
<td></td>
<td>R2(A)</td>
</tr>
<tr>
<td></td>
<td>Rel2(A) -&gt; Last access of obj A</td>
</tr>
</tbody>
</table>

The above schedule would not be equivalent to corresponding serial schedule. (Since order of writes would differ resulting in different values).
b. It can result in deadlock, we can by using the following counter example (Assuming txns know about their future actions):

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>XLock1(A)</td>
<td></td>
</tr>
<tr>
<td>W1(A)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>XLock2(B)</td>
</tr>
<tr>
<td></td>
<td>W2(B)</td>
</tr>
</tbody>
</table>

- XLock1(B) -> Wait1(B), asks for an exclusive lock for B for Write operation but doesn’t get it hence waits for T2 to complete
- XLock2(A) -> WAIT2(A) asks for an exclusive lock for A for Write operation but doesn’t get it hence waits for T1 to complete

- … future another W1(A) therefore does not relinquish the lock
- … future another W2(B) therefore does not relinquish the lock

This results in deadlock.
6. (Solved and Graded by: Pengyu.)
   a. Consistency: satisfied – only the partition with the quorum of servers will be able to execute read/write
      Availability: violated – partition with less than quorum of servers cannot participate in any operation
   b. Consistency: satisfied - as no reads are possible
      Availability: violated - as no read/write will be executed
   c. Consistency: violated – NJ partition can provide stale data
      Availability: violated – NJ partition cannot execute write
   d. Consistency: violated – read from the partition with less than quorum of servers might provide stale data
      Availability: violated - the partition with less than quorum of servers cannot participate in write
   e. Consistency: violated – the value for key K might be different between partition 1 and partition 2
      Availability: depends - on the quorum number of responsive servers
   f. Consistency: satisfied - as no write until partition resolves
      Availability: violated - no write during the partition
   g. Consistency: violated – write on the “less than quorum” partition won’t be visible in the other partition
      Availability: violated - the partition with less than quorum of servers cannot participate in read
   h. Consistency: violated – NJ will not see NYC’s write
      Availability: violated – NYC cannot process reads
7. (Solved and Graded by: Taksh.)
Cloud has 20 CPUs and 50GB RAM
   a. 5 tasks of Job1(10 CPUs, 20GB RAM) and 5 tasks of Job2(10 CPUs, 20GB RAM)
      \[ \max(x, y) \text{ s.t. } 2x + 2y \leq 20 \text{ and } 4x + 4y \leq 50 \text{ and } x=y \]
   b. 3 tasks of Job1(12 CPUs, 9GB RAM) and 4 tasks of Job2(8 CPUs, 32GB RAM)
      \[ \max(x,y) \text{ s.t. } 4x + 2y \leq 20 \text{ and } 3x + 8y \leq 50 \text{ and } 4x/20 = 8y/50 \]
   c. 6 tasks of Job1(12 CPUs, 48GB RAM) and 1 task of Job2(8 CPUs, 1GB RAM)
      \[ \max(x,y) \text{ s.t. } 2x + 8y \leq 20 \text{ and } 8x + y \leq 50 \text{ and } 8x/50 = 8y/20 \]
8. (Solved and Graded by: Samarth.)
   a. Speed and IDs of top 5 fastest whales

   *Data passed along nodes*

   If number of nodes in the subtree <=5, pass the speed and IDs of all sensors.
   Else, pass the speed and IDs of the top 5 fastest nodes in the subtree.

   **Aggregation Function**

   ```python
def aggregate(data_from_children):
    # data_from_children: list of data from all child nodes
    # Each data is a list of (speed, ID)

    combined_data = []
    for data in data_from_children:
        combined_data.append(data)
    sorted_data = sorted(combined_data, reverse=True, key=lambda x:x[0])
    return sorted_data[:5]
```

   b. Count of all whales (currently in the system)

   *Data passed along nodes*

   Count of whales in the subtree

   **Aggregation Function**

   ```python
def aggregate(count_from_children):
    # count_from_children: list of count of whales in each child node
    # for leaf nodes, count_from_children = []
    return sum(count_from_children) + 1
```

c. Average speed across all whales

   *Data passed along nodes*

   Total speed of all whales in subtree, count of whales in subtree, average speed of whales in subtree

   **Aggregation Function**

   ```python
def aggregate(total_speeds, counts, average_speeds):
    # total_speeds: list of sum of speeds in the subtree of child nodes
    # counts: list of count of whales in subtree of child nodes
    # average_speeds: list of average speeds in subtree of child nodes
    totalSpeed = node.speed + sum(total_speeds)
    totalCount = 1 + sum(counts)
    averageSpeed = totalSpeed / totalCount
```
return totalSpeed, totalCount, averageSpeed

d. 75th percentile value of speed across all whales

Data passed along nodes
Speeds of all whales in subtree, 75th percentile of speeds of whales in subtree

Aggregation Function
def aggregate(speeds, percentileSpeeds):
    # speeds: list of list of speeds in subtree of each child node
    # percentileSpeeds: list of 75th percentile speed in subtree of child node
    combinedSpeeds = []
    for speed in speeds:
        combined.append(speed)
    sortedSpeeds = sorted(combinedSpeeds)
    count = len(sortedSpeeds)
    index = int(count * 0.75)
    return sortedSpeeds, sortedSpeeds[index]
9. (Solved and Graded by: Matt.)
   a. Advantages and Disadvantages (not exhaustive)
      i. Advantages
         1. Servers do not need to keep track of or send callback promises for file accesses, lowering the load on the server
         2. If the freshness interval is small, this approach encourages consistency over sending callbacks only after a write is finished
         3. If a client is frequently accessing entire files, this approach is simpler than fetching and caching file blocks
      ii. Disadvantages
         1. If the freshness interval is not small enough, clients are not immediately informed about a write to a file they have cached, leaving room for inconsistency across caches
         2. If the freshness interval is too small, the network will have increased bandwidth relative to that of sending callbacks
   b. Prefetching
      i. Advantages
         1. Given that files tend to be small (relatively), and that there may be spatial locality to files in a directory, this reduces the overhead of future file accesses
      ii. Disadvantages
         1. If we don’t have spatial locality, then we are wasting resources on unnecessary work
   c. Cache Validity
      i. Pending explanation
         1. This could be a good idea since it encourages higher consistency between the client and server, which can be preferable in a write heavy system (i.e. files are quickly becoming stale).
         2. This is a bad idea since it increases the traffic load between the client and the server (checking Tm_client = Tm_server), rather than making an initial local check using the freshness interval.
10. (Solved and Graded by: Tomoko.)
   a. P3 should:
      i. Ask all other processes to invalidate page using multicast
      ii. Mark page as Write mode
      iii. Do write
   b. Incorrect setup: When a page is in Write mode, only the owner has a copy. So, P3 cannot be the owner of the page because P4 has the page in Write mode
   c. P3 should:
      i. Ask all other processes to invalidate their copies of the page using multicast
      ii. Fetch all copies and mark the latest copy as Write mode
      iii. Become the owner of that page
      iv. Do write
   d. Incorrect setup: When a page is in Write mode, only the owner has a copy. So, P1 and P2 cannot hold a page in Write mode if P3 is the owner. This also means P1 and P2 can’t both be in Write mode because there cannot be more than one owner.
   e. Incorrect setup: P1, P2, and P3 cannot all hold the page in Write mode since a page has to be owner to hold a page in Write mode and there can only be one owner.
   f. P3 should:
      i. Ask all other processes to invalidate their copies of the page using multicast
      ii. Mark the page as Write and become the owner
      iii. Do write
   g. P3 should:
      i. Ask all other processes to invalidate their copies of the page using multicast
      ii. Fetch all copies and mark the latest copy as Write mode
      iii. Become the owner of that page
      iv. Do write
   h. Incorrect setup: Every page must have an owner, and two processes cannot both be in Write mode.
   i. Incorrect setup: Every page must have an owner, and P1 should be that owner.
   j. P3 should:
      i. Ask all other processes to invalidate their copies of the page using multicast
      ii. Fetch all copies and mark the latest copy as Write mode
      iii. Become the owner of that page
      iv. Do write

===== END OF HOMEWORK 4 SOLUTIONS =====