1. (Solution and Grading by: Hongwei Wang)
   a. It is not serially equivalent. The order of conflict operations is not consistent. e.g.  
      write(b, caz, T1) and read(b, T2) => T1 before T2.  
      read(c, T2) and write(c, foo, T1) => T2 before T1.  
   b. It is not serially equivalent. The order of conflict operations is not consistent. e.g.  
      read(b, T2) and write(b, caz, T1) => T2 before T1.  
      read(a, T1) and write(a, baz, T2) => T1 before T2.  
   c. It is serially equivalent. The results are equivalent as executing T1 then T2.  
   d. It is not serially equivalent. The order of conflict operations is not consistent. e.g.  
      read(b, T2) and write(b, caz, T1) => T2 before T1.  
      read(a, T1) and write(a, baz, T2) => T1 before T2.  

2. (Solution and Grading by: Xiaocheng Yuan)  
   With this scheme, there cannot be deadlocks regardless of when locks are acquired, at  
   the beginning of transactions or in the middle of transactions.  
   To show that deadlocks cannot happen, we'll prove by contradiction that circular wait is  
   impossible if objects are locked in the same order.  
   Proof:  
   Suppose there is a deadlock currently in the system, then there must be a circular wait  
   among a certain set of transactions. If all transactions were following the described locking  
   scheme which locks objects in a decreasing order, every transaction in the circular wait cycle  
   should be waiting on an object with a lower id than the objects it has already acquired. However,  
   since there is a circular wait cycle, combined with the transitive nature of the lexicographical  
   order, we would derive objects whose ids are both lower and higher than than some other  
   objects(e.g. A < B < C < A), which cannot happen. This shows that it is impossible for circular  
   wait to happen and hence no deadlocks should occur.  

3. (Solution and Grading by: Sili Hui)  
   a. Does not satisfy serial equivalence, T1 acquired lock1 -> T2 acquired lock2 -> T1  
      acquire lock2 when releasing lock1 -> T2 acquire lock1 when releasing lock2.  
   b. There could be deadlock. T1 acquired lock1, when releasing lock1 try to acquire  
      lock2. T2 acquired lock2, when releasing lock2 try to acquire lock1.  

4. (Solution and Grading by: Xiaocheng Yuan)  
   a. Consistency and availability are both violated because writes will not propagate  
      and read is disabled in the NYC partition.  
   b. Consistency and availability are both violated because NJ will not read the latest  
      values and NJ cannot write.
c. Consistency and availability are both violated because the partition without a quorum will not see the latest results, and cannot write.
d. Consistency and availability are both violated because write results on the partition without a quorum cannot be read by the other one, and no reads are allowed on the partition without a quorum.
e. Consistency is violated because read results will differ, but availability is satisfied since all operations are allowed in both partitions.
f. Consistency is satisfied since all reads return the same data, but availability is violated since no writes are allowed.
g. Consistency is satisfied because reads will always return latest results, but availability is violated since one partition cannot do anything.
h. Consistency is satisfied since no reads will be made, but availability is violated since no operations are allowed.

5. (Solution and Grading by: Sili Hui)
a. Samza: support for state, integration with YARN and Kafka, good with large state partitioning, more fault tolerance as state is replicated across machines
b. Storm: expressive graphs (e.g., DAGs) of computation, popular and used by many companies, in many applications, user-defined topology for real-time computation, fault tolerance is done by restarting the worker, adopts Thrift API that allows other languages

6. (Solution and Grading by: Si Liu)
a. Logarithmic
b. Linear
c. Zipf
d. Exponential

7. (Solution and Grading by: Shiv Verma)
a. Advantages and disadvantages of each method:
   i. Method I has regular updates and efficient distribution of load
   ii. Method II has the same advantages of I but doesn’t suffer from a SPoF (the single base station). But complicated as we’ll have to make sure that every mote is present in only one tree at a time.
   iii. Hard to have up-to-date data because of unreliable and seasonal whale swimming patterns. (What if all the whales are far away from the shore?) Also impossible to monitor whales far away from the shore.
b. I’d prefer the 2nd method, as the added complexity isn’t that much and lack of a SPoF increases the fault tolerance a lot. Other answers are accepted if a good reason is provided.
c. You would have to pass along the
   i. Current Running Average
ii. The number of motes that were averaged to get this number (including self)

8. (Solution and Grading by: Bo Teng)
   a. Mark the local copy as “Write”, write to cache
   b. This case is impossible. Only owner can hold the copy in “Write” mode. If the once copy
      is in “Write” mode, no other copies should exit
   c. Use multicast to ask other copies to invalidate their copies (including P4). Fetch the copy
      and mark as “Write”, become owner, then write to copy
   d. This case is impossible. Only owner can hold the copy in “Write” mode. If the once copy
      is in “Write” mode, no other copies should exit
   e. This case is impossible. Only owner can hold the copy in “Write” mode. If the once copy
      is in “Write” mode, no other copies should exit
   f. Use multicast to ask other copies to invalidate their copies (including P2, P4). Mark local
      copy as “Write”, become owner, then write to copy
   g. Use multicast to ask other copies to invalidate their copies (including P2, P4). Fetch all
      copies, use latest one and mark as “Write”, become owner, then write to copy
   h. This case is impossible. Only owner can hold the copy in “Write” mode. If the once copy
      is in “Write” mode, no other copies should exit
   i. P1 must be the owner. Use multicast to ask other copies to invalidate their copies
      (including P1). Fetch the copy and mark as “Write”, become owner, then write to copy
   j. Use multicast to ask other copies to invalidate their copies (including P4, P5). Fetch all
      copies, use latest one and mark as “Write”, become owner, then write to copy

9. (Solution and Grading by: Bo Teng)
   ACLs are the best choice, especially since revocation typically revokes one one object for one
   principal. ACLs are held typically by the server and can be changed quickly. Capabilities may be
   copied by clients and reused, and it's hard to revoke them. ACLs is also more space efficient
   compared to the access control matrix. One disadvantage of using ACLs is if the principal is
   evil and is being completely removed, then all ACLs will need to be visited.

10. (Solution and Grading by: Si Liu)
    a. No. Here are two problems,
    1. Auto advancing read-write pointers: to satisfy at-least once invocation semantics, you
       can’t have auto advancing read-write pointers because it is not idempotent. With at-least
       once invocation semantics, suppose the write operation worked correctly, but the ACK
       reply failed. There will be another write operation, but because of the auto-advancing, it'll
       create an incorrect state for data.
    2. Being stateless: the servers shouldn't maintain per file/ session info, which is not the
       case in UNIX. So, if OS crashes, all the processes have also crashed. This is
       not expected of a distributed file system, where you can have fault tolerance, depending
       on the application use-case.

    b. Move the read-write pointer to the client side, where:
- You can ensure that the read/write functions use absolute positions and remove access of file descriptors.
- As the clients will specify absolute position for their read/write operations, there won't be any need for server to maintain any information. So, if a server fails, another can take over.