The purpose of this problem set is to give you some practice with concepts related to recovery, parallel query processing, two-phase commit, and other papers we read during the second half of the course.

1 Serializability

1. [6 points]: Below, we provide 3 interleavings of several concurrent transactions (consisting of READ and WRITE statements on objects).

Your job is to indicate whether, for each of the interleavings:

– The interleaving has an equivalent serial ordering. If so, indicate what the serial ordering is.
– Whether the interleaving would be permitted using lock-based concurrency control. Assume that locks, when needed, are acquired (with the appropriate lock mode) on an object just before the statement that reads or writes the object and that all locks are released during the COMMIT statement (and no sooner.) If the interleaving is not valid, indicate whether or not it would simply never occur or would result in deadlock, and the time when the deadlock would occur.
– Whether the ordering would be valid using snapshot isolation. If not, indicate which transaction will be aborted. Assume that snapshot isolation is implemented in the same way as optimistic concurrency control in the paper by Kung and Robinson, except that read sets are not track (and read-write / write-read conflicts are ignore.) You can assume that serial validation is used the write phases of optimistic concurrency control happen during the COMMIT statement (and no sooner.)

Assume in all cases that written values can depend on previously read values. (The interleavings are shown on the next page.)
Interleaving 1:
1 T1: READ A
2 T2: READ B
3 T1: WRITE B
4 T2: WRITE A
5 T1: COMMIT
6 T2: COMMIT

Answer: not serializable (T1 Read A) → (T2 write A) implies only the ordering T1 → T2 is possible. But (T2 Read B) → (T1 Write B) implies only T1 → T2 is possible. So this sequence of operations is not serializable. 
not 2PL compatible One way to see this is that the interleaving is not serializable. Also we can see what happens if we run 2PL: T1 blocks on T2 when acquiring the exclusive lock on B, while T2 blocks on T1 while acquiring the exclusive lock on A. SI compatible Because the write sets do not intersect, SI would allow this schedule.

Interleaving 2:
1 T1: READ A
2 T1: READ C
2 T2: WRITE C
3 T2: COMMIT
4 T1: WRITE A
5 T1: COMMIT

Answer: serializable This ordering is conflict-equivalent to T1 → T2. And (T1 Read C) → (T2 Write C) implies this is the only schedule consistent with the interleaving we see. not 2PL compatible. Transaction 2 will need to block when trying to acquire the exclusive lock on C, so this schedule would not happen in 2PL even though it is serializable. SI compatible. The write sets of the two transactions do not intersect, so SI would allow this transaction.

Interleaving 3:
1 T1: READ A
2 T2: READ A
3 T3: READ A
4 T2: WRITE B
5 T2: COMMIT
6 T3: WRITE A
7 T3: COMMIT
8 T1: WRITE A
9 T1: COMMIT

Answer: not serializable If we ignore T2, we see T1 and T3 alone cannot be serialized as we have (T1 Read A) → (T3 Write A) but also (T3 Read A) → (T1 Write A). not 2PL compatible Again, we can see this because the schedule is not serializable. But also, when T3 attempts to acquire exclusive locks on A, it will wait on T1, which will also block on trying to upgrade A to an exclusive lock. not SI compatible Both T1 and T2 have the same write set. On serial validation, T1 will detect that A has changed since it read it earlier on and will abort. T2 and T2 will commit.
2 Recovery

2. [2 points]: If your DBMS always FORCED pages (as discussed in class and in the paper by Franklin), how would that affect the design of the database recovery manager? Answer: A FORCE system will result in all writes being on disk prior to transaction commit. This means that there will never be a need to REDO any committed transactions. Thus, the recovery/logging system will not need to record the READ (after) image for any page.

3. [3 points]: Suppose you are told that the following transactions are run concurrently on a database system that has just been restarted and is fully recovered, and is running Rigorous 2PL. Suppose the system crashes while executing the statement marked by an “***” in Transaction 1. Suppose that Transaction 2 has committed, that the write of Z in T4 has been flushed to the database, and the state of Transactions 3 and 4 are unknown (e.g., they may or may not have committed.) Assume that each object (e.g., X, Y, etc.) occupies exactly one page of memory.

T1 T2 T3 T4
-- -- -- --
RX RY RZ RX
***WX RZ WZ RZ
RV WY RW WZ
WV WW

A. Show an equivalent serial order that could have resulted from these statements, given what you know about what statement was executing when the system crashed. In addition, show an interleaving of the statements from these transactions that is equivalent to your serial order; make sure this serial order could result from a rigorous two-phase locking-based concurrency control protocol.

Answer: There are many correct answers here. T2 must have run before T1. T4 must have committed because its write of Z is on disk, and it conflicts with T1. T3 could have run to completion before T2, after T2, or not run at all.

One possible schedule is T2, T4, T1, T3.
It's simplest to just run these serially, e.g.,:

T2: RX
T2: RZ
T2: WY
T4: RX
T4: RZ
T4: WZ
T1: RX
T1: WX
T1: RV
T1: WY
T3: RZ
T3: WZ
T3: RW
T3: WW

B. Show all of the records that should be in the log at the time of the crash (given your serial order), assuming that there have been no checkpoints and that you are using an ARIES-style logging and recovery protocol. Your records should include all of the relevant fields described in Section 4.1 of the ARIES paper. Also show the status of the transaction table (as described in Section 4.3 of the ARIES paper) after the analysis phase of recovery has run.
C. Suppose you have 2 pages of memory, and are using a STEAL/NO-FORCE buffer management policy as in ARIES. Given the interleaving you showed above, for each of the 5 pages used in these transactions, show one possible assignment of LSN values for those pages as they are on disk before recovery runs. You should use the value “?” if the LSN is unchanged from the prior state of the page before these transactions began. Finally, indicate which pages will be modified during the UNDO pass, and which will be modified during the REDO pass.

<table>
<thead>
<tr>
<th>Page</th>
<th>LSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>?</td>
</tr>
<tr>
<td>W</td>
<td>?</td>
</tr>
<tr>
<td>X</td>
<td>?</td>
</tr>
<tr>
<td>Y</td>
<td>2</td>
</tr>
<tr>
<td>Z</td>
<td>4</td>
</tr>
</tbody>
</table>

Answer:
3 Parallel Query Processing

The standard way to execute a query in a shared-nothing parallel database is to horizontally partition all tables across all nodes in the database. Under such a setting, distributed joins can be computed by repartitioning tables over the join attributes.

Suppose these tables are partitioned across a 4 node distributed database, where each node has 1 TB of disk storage and 10 GB of RAM. Also, assume that:

- The disk can read at 50 MB/sec (for the purposes of this problem, you may ignore differences between sequential and random I/O),
- The network on each node can transmit data at 40 MB/sec, regardless of the number or rate at which other nodes are simultaneously transmitting,
- A computer cannot send over the network and read or write from its disk at the same time,
- CPU costs are negligible relative to disk or network I/O time.

For each of the following queries, indicate what partitioning strategy you would recommend. This is the only query run by the system, and your job is to choose the best up-front partitioning for this particular query / setting. Choose between replicating each table on all nodes, or hash, range, or round-robin partitioning it. For hash partitioning, specify the attribute you would use; for range partitioning specify the ranges you would use.

4. [3 points]:

```
SELECT COUNT(*)
FROM R JOIN S
ON R.a = S.b // S.b is a primary key
WHERE R.a = 1
```

Here, R.a is a primary key, R is 2 TBs and several billion records, and S is 100 MB and several million records.

Indicate the best partitioning / replication strategy (and partitioning attributes) for R and S, as well as a brief explanation.

Answer: For R, we choose round-robin partitioning on R.a. The reason is that if we partition by hash or range, the query will always hit a single node and we will not benefit from parallelism (particularly if R.a = 1 has low selectivity). Since we partition R using round-robin, we replicate S on all nodes.

5. [3 points]:

```
SELECT COUNT(*)
FROM R JOIN S
ON R.a = S.b // S.b is a primary key
WHERE R.a > 100 // this predicate has a selectivity of .1
```

Here, R.a and S.b are primary keys, R is 2 TBs and several billion records, and S is 1 TB and several hundred million records.

Indicate the best partitioning / replication strategy (and partitioning attributes) for R and S, as well as a brief explanation.

Answer: Note that the selection predicate R.a has relatively high selectivity. Since R and S both have sizes in TB, we need to partition both either using hash or range partitioning. If we use range partitioning, it is unclear what ranges will ensure a good distribution across all nodes (e.g. if we use range partitioning such that all records with R.a > 100 are on the same node, then we must partition S.b similarly but we have no idea what the selectivity of S.b > 100 will be) Instead if we use hash partitioning on R.a and S.b, we expect the load to be distributed across all nodes.
6. [3 points]:

SELECT R.a, COUNT(*)
FROM R
WHERE R.b > 100 //this predicate has selectivity of .001
GROUP BY R.a //there are 10000 groups

Here, R is 2 TBs and several billion records.

Indicate the best partitioning / replication strategy (and partitioning attributes) for R, as well as a brief explanation.

Answer: Partition R on R.a via hash partitioning. This query can be executed by applying the filter and aggregation while performing a single pass through the data. We use hash partitioning to reduce the cost of the merge at the coordinator node.
4 Tweeter

Dana Bass is building a instant messaging system called Tweeter. In Tweeter, messages can be addressed to one or more other users. Tweeter is supposed to provide the following transactional guarantees about messages:

- **All or nothing**: When a message is sent, it is delivered to all of its recipients, or none of them; if one message cannot be delivered (because, for example, the machine that is supposed to store it is unavailable), none of the recipients should receive it.
- **In order**: If user U1 and U2 both receive messages M1 and M2, then if U1 sees M1 before M2, U2 should also see M1 before M2.

Dana implements Tweeter as a three node distributed database. Each user is given a separate table for his or her incoming messages, and each user's messages table is stored on one of the nodes. When a message is sent, it is written into the messages table of each user who receives it.

7. [8 points]: Initially, to ensure that Tweeter provides the all-or-nothing property, Dana uses a database system that uses two-phase commit (in the standard form, without presumed abort or presumed commit), and writes each message as a part of a single transaction. Even though her machines are on a fast LAN with network round trip times of only 100 ns, she finds that it takes about 20 ms to commit a transaction for a message, even when only one transaction is running at a time. Explain, in one sentence, what is likely going on.

**Answer:** Two phase commit requires several forced log writes (a prepare message on each worker, followed by a commit on the coordinator.) Each of these probably takes about 10 ms.
To improve the performance of Tweeter, Ben Bitdiddle tells Dana that because messages are never deleted or updated, she can use a much simpler protocol than two-phase commit. Ben tells Dana about the Network Time Protocol (NTP), which can ensure that nodes in a cluster have clocks that are synchronized (i.e., agree with each other) to within 1 millisecond. He proposes Dana run NTP on her machines and then use the following protocol for sending a message:

- To send a message $m$, a client connects to one of the nodes $N$. The node assigns the message a 96-bit commit timestamp $TS_m$. To compute a timestamp, a node appends its unique 32-bit nodeid onto a 64-bit time in milliseconds since 1/1/2000 (the high order bits of the timestamp are the time); a node never issues the same timestamp for two messages.
- $N$ sends $m$ to each of the nodes that store the table for a user who should receive $m$, along with the value $TS_m$.
- When a node receives a message $m$ for a user $u$, it waits 2 milliseconds. If no messages arrive with a timestamp less than $TS_m$, the node “posts” $m$ by running a local transaction to write the message to $u$’s table. Otherwise, it posts the messages with timestamps less than $TS_m$ and then posts $m$.

8. [8 points]: Assuming that nodes can crash at any time, but that the network is never partitioned, delivers all messages, and never takes more than 100 ns to send messages between two nodes, does Ben’s protocol preserve the all-or-nothing property of Tweeter? Why or why not?

Answer: No. If a node receives a message and then crashes, it may lose the message while other nodes may display it, since there’s no logging in Ben’s protocol.

The protocol above does preserve the in order property. However, if it is modified such that nodes only wait 0.5 ms instead of 2 ms, the in order property can be violated.

9. [8 points]: Given an example of a sequence of messages and timestamps that could result in a violation of the Tweeter in order property when nodes only wait 0.5 ms.

Answer: There are several possible ways in which Ben’s protocol can fail. Here is one: suppose the following sequence of events (note that .5 ms is 500,000 ns):

100 ns N2 produces message $m_2$ with timestamp $\{0.2\}$ (this notation means a message with timestamp 0 milliseconds was produced by node 2) for nodes $\{N1, N2\}$ and sends it to N1.

200 ns $m_2$ arrives at N1

500,050 ns N1 produces message $m_1$ with timestamp $\{0,1\}$ for $\{N1, N2\}$ and sends it to N2

500,100 ns N2 posts message $m_2$ after waiting 500,000 ns (.5 ms). This is before it has received message $m_1$ which should be ordered earlier because the messages have the same time value and N1 < N2.

500,150 ns $m_1$ arrives at N2

500,200 ns N1 posts message $m_1$ before message $m_2$, since after N1 receives $m_2$, it starts waiting and receives message $m_1$ at time 500,050 ns. It therefore (correctly) posts $m_1$ before $m_2$.

The in-order property is violated because the nodes post the messages in different orders. This is illustrated in the following diagram:
Note that with a 2 ms timeout this problem does not arise, because if two nodes timestamped messages in the same millisecond, and messages take 100 ns to propagate, each node will definitely hear the message from the other node before posting its own message (and the way we have constructed timestamps ensures nodes will agree on the order in which messages should be posted).

Other failure cases can arise if you assume three nodes, with one node sending the same message to two receivers, and the message taking different amounts of time to reach each receiver. Interestingly, the inaccuracy of time synchronization between the nodes does not matter in either case.