Assignment 4
Due: May 24, 2022

Question 1: Assume that a transaction T consists of the following operations, resulting from hiring a new manager 'Jill' for the 'toy' department of a company

T: **Begin**

insert into Employee values (<Jill, 120k>)

Update Departments set manager_name = 'jill" where department_name = 'toy'

Commit

Let the file containing Employee and Departments table to be named Employee and Departments respectively. Also, there is a B-tree index on the Salary field for Employee and also on the name field. Finally, there is a B-tree on manager_name and department_name of Departments. Let the files corresponding to these B-trees be name, sal, mname, and dname respectively. The insertion of the Employee table causes insertion on page 7 of Employee file, and insertion into the corresponding B-trees causes updates to pages 9 and 11 of name file and pages 13, 15, and 17 of sal file. Furthermore, update of Departments table causes page 19 of the Departments file to be updated and an insertion of the record on page 21 of the mname file corresponding to the B-tree. Let us assume that all the pages of all files are initially in disk.

Let us assume that we are using logical logging with logical undo, and, furthermore, using shadow paging as a mechanism for ensuring action consistency of logical operators. Sketch out the exactly how transaction processing will proceed, what logs will be written, what locks will be acquired, when, and when will they be released. Remember, the goal is to maximize concurrency, so the page locks should be held for as short a duration as possible. Note that the problem is a bit underspecified in the sense that it does not specify where the page tables corresponding to each of the files is stored and where the master page table is stored. Feel free to "fill in the gaps" in the question while answering it. Answer the question under the assumption that the transaction T above requests commit at the end of the transaction.

Question 2: Let us redo the question above, but this time instead of shadow paging for action consistency, we rather do physical logging. Physical logging requires you to know the before and after value of the part of the page that is changed. You can simply refer to the before and after values as b(P,O) and a(P,O) where P is the page and O is the operation.

This time when you explain how the transaction processing proceeds, assume that the transaction aborts mid flight after successfully completing the insert operation during the execution of the update. In particular it fails after inserting the tuple on Departments file on page 19, but just before the corresponding insertion on page 21 of the mname B-tree.

Question 3: Consider an ARIES style algorithm that uses pageLSN to track state of the pages, and CLRs are written to ensure that the page state is tracked correctly irrespective of rollbacks in the middle of the execution.

1. Explain the importance of writing CLRs. One option in undoing log record with lsn l_1, instead of writing CLRs during transaction roll back, is to reset the pageLSN of the data page to the LSN of the log record that modified the page before l_1 (that is, the log record with a lower lsn value).Construct a counterexample that illustrates what can go wrong if this scheme is used for handling transaction rollbacks.
2. Assume that we were using page level locks to ensure strictness. Is writing CLRs still important? That is, can you design transaction roll back and restart algorithms that work correctly but do not require CLRs to be written under the assumption of page level locking.
3. In the restart algorithm, during the redo pass we repeat the entire history (that is, the effects of both committed as well as transactions that were active at the time of failure). One alternative is to instead perform redo selectively. That is, we only redo the effects of committed transactions and not of those transactions that were active at the time of failure. Illustrate through a counterexample what can go wrong with our algorithm if we were to perform selective redos.

Question 4: The force log at commit requirement for transactions dictates that a transaction's log records need to be forced to disk before it commits. This, however, implies that a disk write must take place for every transaction commit. Let us assume that writing a page to disk (once the disk head has been properly aligned) takes 20 milliseconds (At disk transfer rate of approx. 1MBs, the time to write a 10KB page to disk is 10/1000s = 10millisecods. Furthermore, each disk write has a fixed overhead of dispatching the daemon, preparing and issuing the I/O, and cleaning up after the I/O completes. This overhead could be approx. 10 milliseconds. Thus the total time to write on disk is approx. 20 milliseconds) which imposes a bound on the maximum throughput of 50 tps. To improve on throughput, a technique developed by IMS/ Fast Path developers was the idea of *group commit* In IMS fast path, the log records of the transaction (including its commit record) are not forced to disk immediately after the completion of the transaction. Instead, the log manager waits for the log page to fill up. By writing only full log pages, and writing as many pages as possible in a single write, the system is able utilize full disk bandwidth and to amortize the fixed software overhead of writing to disk over many transactions. In the group commit scheme, the log daemon wakes up, say every 100 millisecond, and flushes all the log pages which are in buffer to disk. This way the fixed software overhead of 10 millisecond is paid every 100 milliseconds and thus 90 out of every 100 milliseconds can be used for transferring data to the disk. That is, the system can utilize .9 of the disk bandwidth thereby being able to write .9MB/second of log data to the disk. Assuming that each transaction, on average, writes 1KB of log data, the group commit optimization potentially raises the system throughput to approximately 900 transactions per second. One problem with the group commitment, however, is that since it batches the commitment of transactions, it increases the average response time of transactions 50 milliseconds since a transaction, even though it has finished execution, is not committed until the log daemon wakes up and pushes the transaction's commit log record to the stable storage. While an increase of 50 milliseconds in the response time for transactions is not a big problem, unfortunately, since strictness dictates that the locks held by transactions not be released until the transaction commits, the group commit strategy increases the lock hold time of transactions by 50 milliseconds. Recall from the discussion in class that increase in lock hold time increases the conflict and deadlocks in the system thereby resulting in decrease in throughput. To overcome this problem, the approach taken by the IMS designers is to release the locks held by the transaction when it completes execution even before its log records have been stably recorded to the disk. Notice that this is a violation of strictness of schedules. In fact, since after the lock release, other transactions can read the data written by a transaction even before the transaction commits, the resulting schedule is not even cascadeless.

1. How will the recovery algorithm we studied in class have to be modified to work correctly with the early lock release which is used in conjunction with group commit as we described above. Answer this question under two different sets of assumptions about the

system. *(Assume that we have a centralized database with a single log file and logs are flushed to disk sequentially in an order in which log records are written).*

2. Now consider a distributed system where each site has an independent log manager that exploits group commit and consequently locks at the sites are released when the transaction is ready to commit (that is, on receipt of the prepare message if a site is a cohort or when the commit decision occurs if the site is a coordinator). Can you design a recovery approach that still works correctly in this case?

Question 5: Suppose that a fuzzy checkpointing scheme is used and at the time of the checkpoint all the dirty pages are written to the disk.For each checkpoint, a begin_checkpoint and end_checkpoint log records are written. Furthermore, suppose that the recovery algorithm uses only a single centralized log. Each log record in the log has with it associated a log sequence number (lsn) and a log is a sequential sorted file, sorted by the lsn values. Consider a system failure. Let the lsns of the begin_checkpoint and end_checkpoint log records of the last checkpoint before failure be Ci.lsn1 and Ci.lsn2 respectively. Let the lsns of the begin_checkpoint and end_checkpoint log records of the second last checkpoint be Ci-1.lsn1 and Ci-1.lsn2 respectively. Furthermore, let T1, T2, ..., Tn be the transaction that were active at the time of failure (that is, for whom neither a commit, nor an abort log record had been written at the time of failure) and let Ti.lsn be the log sequence number of the first log record for transaction Ti, i=1,2, .., n. Assume that, as discussed in class, restart consists of three phases, an ANALYSIS PHASE in which we determine the set of transactions that must be rolled back, as well as the set of transactions whose effects must be committed to the disk. A REDO PHASE in which we repeat the history, redoing the effects of the transactions that need to be committed as well as those which need to be aborted. Finally, an UNDO PHASE in which the effects of transactions that need to be rolled back are undone.

1. What is the bound on the lsn of the log record below which we do not need to redo during restart?
2. What is the bound on the lsn of the log record below which we do not need to undo during restart?

Assume that to determine the status of the various transactions before failure, the restart algorithm starts from the last checkpoint record. It traverses the log forward and constructs a *commit*, *abort*, and *active* lists. Initially, these lists are empty. When the restart algorithm sees an update log record for a transaction, it adds the transaction to the *active* list. On the other hand, if it sees a commit log record, it inserts the transaction into the *commit* list and deletes the transaction from the active list(in case the transaction is present in the active list). If it sees an *abort* log record for the transaction, it inserts the transaction to the *abort* list and deletes the transaction from the active list (in case the transaction is present in the active list). Once the restart algorithm sees the last log record it terminates its analysis pass.

1. Is every transaction that was active prior to the system failure present in the {\it active list} at the end of the analysis pass? (Yes/No). Please explain. Note this is not an midterm or final, so you could be a bit verbose in your explanations
2. Is every transaction that had committed prior to the system failure but whose effects may need to be redone using log records present in the commit list at the end of the analysis pass? (Yes/No). Please explain

Question 6:
Paper reading assignment: Read through the survey paper on database recovery techniques. Write a summary of the paper. In your summary highlight the following:
1. Recovery techniques discussed in the paper with a brief description of their novelty.
2. Comparison strategies used to compare different databases in the paper and discuss briefly if any other metrics could have been used to compare them.
3. Briefly mention what you liked and did not like in the paper. Be sure to justify your opinions.