

Question 1: In class we argued that logical logging is attractive since it results in lesser log data. Unfortunately, using logical logging introduces the problem of action consistency and partial action.

- Using an example, explain the action consistency and partial action problems that arise if logical logging is used
- Do these problems also arise if physiological logging is used. If so, how do the recovery algorithms based on physiological logging deal with these problems.
- Do these problem arise in case physical logging is used. Again, if so, how do the solutions based on physical logging deal with these issues.

Question 2: Consider a recovery algorithms (that is, transaction rollback and restart recovery) based on physiological logging we studied in class. In the algorithm, a pageLSN is stored on data pages to trace the state of the page with respect to the log in case of a failure. This is done to ensure idempotence of the undo and the redo processing; that is, to ensure that the restart does not:

- undo an operation if its effects do not appear on the page.
- redo an operation if its effects already appear on the page.

In the algorithm described, to process a write operation w_i on page p_i , the pageLSN value of p_i is modified to the value of the LSN of the corresponding log record of w_i . To process a transaction rollback in case of an abort, for each undo operation, a compensation log record (CLR) is written, and the value of the pageLSN modified to the LSN of the CLR.

Assume that during the checkpoint, we force all the data pages in main memory to stable storage. Thus, during restart, processing begins from the last checkpoint record and all the log records after the checkpoint record are redone (that is, the entire history since the last checkpoint is repeated). After the redo pass is over, an undo pass begins which undoes the effects of the transactions that were not committed before the failure occurred. Undo pass results in writing of CLRs for each undone log record.

After the undo pass, the system takes a checkpoint and the transaction processing begins.

In the following assume that strictness is ensured at the level of records unless otherwise specified.

- What is the importance of writing CLRs during transaction rollback?
- 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 roll backs.

- 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 works correctly but do not require CLRs to be written under the assumption of page level locking.
- 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 counter example what can go wrong with our algorithm if we were to perform selective redos.
- Can you suggest a modification to the restart algorithm which will correctly perform recovery but which does not redo the log records of aborted transactions.
- In the recovery protocol described in the class, during undo pass of the restart, the log records of a transaction that did not commit are undone in reverse chronological order and appropriate CLRs written. Illustrate why this simple strategy may result in unbounded logging in presence of repeated failures. Suggest a mechanism which can be used to ensure bounded amount of logging.

Question 3: 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. 1MB/s, the time to write a 10KB page to disk is $10/1000s = 10\text{milliseconds}$). 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. 10milliseconds. 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 an 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 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.

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 assumption about the system.

- Assume that we have a centralized database with a single log file and logs are flushed to disk sequentially in a order in which log records are written.
- 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 4: 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 sequencenumber (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 $C_i.lsn1$ and $C_i.lsn2$ respectively. Let the lsn of the {\it begin_checkpoint} and {\it end_checkpoint} log records of the second last checkpoint be $C_{i-1}.lsn1$ and $C_{i-1}.lsn2$ respectively. Furthermore, let T_1, T_2, \dots, T_n be the transactions 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 $T_i.lsn$ be the log sequence number of the first log record for transaction T_i , $i=1,2,\dots,n$.

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.

- What is the bound on the lsn of the log record below which we do not need to redo during restart?
- 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 status of the various transactions before failure, the restart algorithm starts from the last checkpoint record. It traverses the log forward and constructs a {\it commit}, {\it abort}, and {\it 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 {\it active} list. On the other hand, if it sees a commit log record, it inserts the transaction into the {\it commit} list and deletes the transaction from the active list (in case the transaction is present in the active list). If it sees an {\it abort} log record for the transaction, it inserts the transaction to the {\it 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.
 - o 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 ☺
 - o 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 {\it commit list} at the end of the analysis pass? (Yes/No). Please explain.