CHAPTER 16

Recovery System

Practice Exercises

16.1 Explain why log records for transactions on the undo-list must be processed in reverse order, whereas redo is performed in a forward direction.
Answer: Within a single transaction in undo-list, suppose a data item is updated more than once, say from 1 to 2, and then from 2 to 3. If the undo log records are processed in forward order, the final value of the data item would be incorrectly set to 2, whereas by processing them in reverse order, the value is set to 1. The same logic also holds for data items updated by more than one transaction on undo-list.
Using the same example as above, but assuming the transaction committed, it is easy to see that if redo processing processes the records in forward order, the final value is set correctly to 3, but if done in reverse order, the final value would be set incorrectly to 2.

16.2 Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed? How does the frequency of checkpoints affect:

- System performance when no failure occurs?
- The time it takes to recover from a system crash?
- The time it takes to recover from a media (disk) failure?

Answer: Checkpointing is done with log-based recovery schemes to reduce the time required for recovery after a crash. If there is no checkpointing, then the entire log must be searched after a crash, and all transactions undone/redone from the log. If checkpointing had been performed, then most of the log-records prior to the checkpoint can be ignored at the time of recovery.
Another reason to perform checkpoints is to clear log-records from stable storage as it gets full.
Since checkpoints cause some loss in performance while they are being taken, their frequency should be reduced if fast recovery is not critical. If we need fast recovery checkpointing frequency should be increased. If

the amount of stable storage available is less, frequent checkpointing is unavoidable.
Checkpoints have no effect on recovery from a disk crash; archival dumps are the equivalent of checkpoints for recovery from disk crashes.

16.3 Some database systems allow the administrator to choose between two forms of logging: *normal logging*, used to recover from system crashes, and *archival logging*, used to recover from media (disk) failure. When can a log record be deleted, in each of these cases, using the recovery algorithm of Section 16.4?
Answer: **Normal logging**: The following log records cannot be deleted, since they may be required for recovery:

a. Any log record corresponding to a transaction which was active during the most recent checkpoint (i.e. which is part of the <checkpoint L> entry)

b. Any log record corresponding to transactions started after the recent checkpoint.

All other log records can be deleted. After each checkpoint, more records become candidates for deletion as per the above rule.
Deleting a log record while retaining an earlier log record would result in gaps in the log, and would require more complex log processing. Therefore in practise, systems find a point in the log such that all earlier log records can be deleted, and delete that part of the log. Often, the log is broken up into multiple files, and a file is deleted when all log records in the file can be deleted.

Archival logging: Archival logging retains log records that may be needed for recovery from media failure (such as disk crashes). Archival dumps are the equivalent of checkpoints for recovery from media failure. The rules for deletion above can be used for archival logs, but based on the last archival dump instead of the last checkpoint. The frequency of archival dumps would be lesser than checkpointing, since a lot of data has to be written. Thus more log records would need to be retained with archival logging.

16.4 Describe how to modify the recovery algorithm of Section 16.4 to implement savepoints, and to perform rollback to a savepoint. (Savepoints are described in Section 16.8.3.)
Answer: A savepoint can be performed as follows:

a. Output onto stable storage all log records for that transaction which are currently in main memory.

b. Output onto stable storage a log record of the form <savepoint T_i>, where T_I is the transaction identifier.

To roll back a currently executing transaction partially till a particular savepoint, execute undo processing for that transaction, till the savepoint is reached. Redo log records are generated as usual during the undo phase above.
It is possible to perform repeated undo to a single savepoint by writing a fresh savepoint record after rolling back to that savepoint. The above algorithm can be extended to support multiple savepoints for a single transaction, by giving each savepoint a name. However, once undo has rolled back past a savepoint, it is no longer possible to undo upto that savepoint.

16.5 Suppose the deferred modification technique is used in a database.

a. Is the old-value part of an update log record required any more? Why or why not?

b. If old values are not stored in update log records, transaction undo is clearly not feasible. How would the redo-phase of recovery have to be modified as a result?

c. Deferred modification can be implemented by keeping updated data items in local memory of transactions, and reading data items that have not been updated directly from the database buffer. Suggest how to efficiently implement a data item read, ensuring that a transaction sees its own updates.

d. What problem would arise with the above technique, if transactions perform a large number of updates?

Answer:

a. The old-value part of an update log record is not required. If the transaction has committed, then the old value is no longer necessary as there would be no need to undo the transaction. And if the transaction was active when the system crashed, the old values are still safe in the stable storage as they haven't been modified yet.

b. During the redo phase, the undo list need not be maintained any more, since the stable storage does not reflect updates due to any uncommitted transaction.

c. A data item read will first issue a read request on the local memory of the transaction. If it is found there, it is returned. Otherwise, the item is loaded from the database buffer into the local memory of the transaction and then returned.

d. If a single transaction performs a large number of updates, there is a possibility of the transaction running out of memory to store the local copies of the data items.

16.6 The shadow-paging scheme requires the page table to be copied. Suppose the page table is represented as a B$^+$-tree.

a. Suggest how to share as many nodes as possible between the new copy and the shadow-copy of the B$^+$-tree, assuming that updates are made only to leaf entries, with no insertions and deletions.

b. Even with the above optimization, logging is much cheaper than a shadow-copy scheme, for transactions that perform small updates. Explain why.

Answer:

a. To begin with, we start with the copy of just the root node pointing to the shadow-copy. As modifications are made, the leaf entry where the modification is made and all the nodes in the path from that leaf node till the root, are copied and updated. All other nodes are shared.

b. For transactions that perform small updates, the shadow-paging scheme, would copy multiple pages for a single update, even with the above optimization. Logging, on the other hand just requires small records to be created for every update; the log records are physically together in one page or a few pages, and thus only a few log page I/O operations are required to commit a transaction. Furthermore, the log pages written out across subsequent transaction commits are likely to be adjacent physically on disk, minimizng disk arm movement.

16.7 Suppose we (incorrectly) modify the recovery algorithm of Section 16.4 to not log actions taken during transaction rollback. When recovering from a system crash, transactions that were rolled back earlier would then be included in undo-list, and rolled back again. Give an example to show how actions taken during the undo phase of recovery could result in an incorrect database state. (Hint: Consider a data item updated by an aborted transaction, and then updated by a transaction that commits.)
Answer: Consider the following log records generated with the (incorrectly) modified recovery algorithm:

1. <T_1 start>
2. <T_1, A, 1000, 900>
3. <T_2 start>
4. <T_2, A, 1000, 2000>
5. <T_2 commit>

A rollback actually happened between steps 2 and 3, but there are no log records reflecting the same. Now, this log data is processed by the recovery algorithm. At the end of the redo phase, T_1 would get added to the undo-list, and the value of A would be 2000. During the undo phase,

since T_1 is present in the undo-list, the recovery algorithm does an undo of statement 2 and A takes the value 1000. The update made by T_2, though commited, is lost.
The correct sequence of logs is as follows:

1. <T_1 start>
2. <T_1, A, 1000, 900>
3. <T_1, A, 1000>
4. <T_1 abort>
5. <T_2 start>
6. <T_2, A, 1000, 2000>
7. <T_2 commit>

This would make sure that T_1 would not get added to the undo-list after the redo phase.

16.8 Disk space allocated to a file as a result of a transaction should not be released even if the transaction is rolled back. Explain why, and explain how ARIES ensures that such actions are not rolled back.
Answer: If a transaction allocates a page to a relation, even if the transaction is rolled back, the page allocation should not be undone because other transactions may have stored records in the same page. Such operations that should not be undone are called nested top actions in ARIES. They can be modeled as operations whose undo action does nothing. In ARIES such operations are implemented by creating a dummy CLR whose UndoNextLSN is set such that the transaction rollback skips the log records generated by the operation.

16.9 Suppose a transaction deletes a record, and the free space generated thus is allocated to a record inserted by another transaction, even before the first transaction commits.

a. What problem can occur if the first transaction needs to be rolled back?

b. Would this problem be an issue if page-level locking is used instead of tuple-level locking?

c. Suggest how to solve this problem while supporting tuple-level locking, by logging post-commit actions in special log records, and executing them after commit. Make sure your scheme ensures that such actions are performed exactly once.

Answer:

a. If the first transaction needs to be rolled back, the tuple deleted by that transaction will have to be restored. If undo is performed in the usual physical manner using the old values of data items, the space allocated to the new tuple would get overwritten by the transaction

undo, damaging the new tuples, and associated data structures on the disk block. This means that a logical undo operation has to be performed i.e., an insert has to be performed to undo the delete, which complicates recovery.
On related note, if the second transaction inserts with the same key, integrity constraints might be violated on rollback.

b. If page level locking is used, the free space generated by the first transaction is not allocated to another transaction till the first one commits. So this problem will not be an issue if page level locking is used.

c. The problem can be solved by deferring freeing of space till after the transaction commits. To ensure that space will be freed even if there is a system crash immediately after commit, the commit log record can be modified to contain information about freeing of space (and other similar operations) which must be performed after commit. The execution of these operations can be performed as a transaction and log records generated, following by a post-commit log record which indicates that post commit processing has been completed for the transaction.
During recovery, if a commit log record is found with post-commit actions, but no post-commit log record is found, the effects of any partial execution of post-commit operations are rolled back during recovery, and the post commit operations are reexecuted at the end of recovery. If the post-commit log record is found, the post-commit actions are not reexecuted. Thus, the actions are guaranteed to be executed exactly once.
The problem of clashes on primary key values can be solved by holding key-level locks so that no other transaction can use the key till the first transaction completes.

16.10 Explain the reasons why recovery of interactive transactions is more difficult to deal with than is recovery of batch transactions. Is there a simple way to deal with this difficulty? (Hint: Consider an automatic teller machine transaction in which cash is withdrawn.)
Answer: Interactive transactions are more difficult to recover from than batch transactions because some actions may be irrevocable. For example, an output (write) statement may have fired a missile, or caused a bank machine to give money to a customer. The best way to deal with this is to try to do all output statements at the end of the transaction. That way if the transaction aborts in the middle, no harm will be have been done.
Output operations should ideally be done atomically; for example, ATM machines often count out notes, and deliver all the notes together instead of delivering notes one-at-a-time. If output operations cannot be done atomically, a physical log of output operations, such as a disk log of events, or even a video log of what happened in the physical world can be

maintained, to allow perform recovery to be performed manually later, for example by crediting cash back to a customers account.

16.11 Sometimes a transaction has to be undone after it has committed because it was erroneously executed, for example because of erroneous input by a bank teller.

a. Give an example to show that using the normal transaction undo mechanism to undo such a transaction could lead to an inconsistent state.

b. One way to handle this situation is to bring the whole database to a state prior to the commit of the erroneous transaction (called *point-in-time* recovery). Transactions that committed later have their effects rolled back with this scheme.
   Suggest a modification to the recovery algorithm of Section 16.4 to implement point-in-time recovery using database dumps.

c. Later nonerroneous transactions can be re-executed logically, if the updates are available in the form of SQL but cannot be re-executed using their log records. Why?

Answer:

a. Consider the a bank account A with balance \$100. Consider two transactions T_1 and T_2 each depositing \$10 in the account. Thus the balance would be \$120 after both these transactions are executed. Let the transactions execute in sequence: T_1 first and then T_2. The log records corresponding to the updates of A by transactions T_1 and T_2 would be $< T_1, A, 100, 110 >$ and $< T_2, A, 110, 120 >$ resp.
   Say, we wish to undo transaction T_1. The normal transaction undo mechanism will replaces the value in question—A in this example—by the old-value field in the log record. Thus if we undo transaction T_1 using the normal transaction undo mechanism the resulting balance would be \$100 and we would, in effect, undo both transactions, whereas we intend to undo only transaction T_1.

b. Let the erroneous transaction be T_e.

   - Identify the latest archival dump, say D, before the log record $< T_e, START>$. Restore the database using the dump.
   - Redo all log records starting from the dump D till the log record $< T_e, COMMIT>$. Some transaction—apart from transaction T_e —would be active at the commit time of transaction T_e. Let S_1 be the set of such transactions.
   - Rollback T_e and the transactions in the set S_1. This completes point-in-time recovery.
     In case logical redo is possible, later transactions can be rexecuted logically assuming log records containing logical redo

information were written for every transaction. To perform logical redo of later transactions, scan the log further starting from the log record $< T_e$, *COMMIT*$>$ till the end of the log. Note the transactions that were started after the commit point of T_e. Let the set of such transactions be S_2. Re-execute the transactions in set S_1 and S_2 logically.

c. Consider again an example from the first item. Let us assume that both transactions are undone and the balance is reverted back to the original value \$100.

Now we wish to redo transaction T_2. If we redo the log record $< T_2, A, 110, 120 >$ corresponding to transaction T_2 the balance would become \$120 and we would, in effect, redo both transactions, whereas we intend to redo only transaction T_2.