CHAPTER 17

Recovery System

Exercises

17.3 Answer:

- The recovery scheme using a log with deferred updates has the following advantages over the recovery scheme with immediate updates:
  - **a.** The scheme is easier and simpler to implement since fewer operations and routines are needed, i.e., no UNDO.
  - **b.** The scheme requires less overhead since no extra I/O operations need to be done until commit time (log records can be kept in memory the entire time).
  - **c.** Since the old values of data do not have to be present in the log-records, this scheme requires less log storage space.
- The disadvantages of the deferred modification scheme are :
  - **a.** When a data item needs to accessed, the transaction can no longer directly read the correct page from the database buffer, because a previous write by the same transaction to the same data item may not have been propagated to the database yet. It might have updated a local copy of the data item and deferred the actual database modification. Therefore finding the correct version of a data item becomes more expensive.
  - **b.** This scheme allows less concurrency than the recovery scheme with immediate updates. This is because write-locks are held by transactions till commit time.
  - **c.** For long transaction with many updates, the memory space occupied by log records and local copies of data items may become too high.

17.6 Answer: The first phase of recovery is to undo the changes done by the failed transactions, so that all data items which have been modified by them get back

the values they had before the *first* of the failed transactions started. If several of the failed transactions had modified the same data item, forward processing of log-records for undo-list transactions would make the data item get the value which it had before the *last* failed transaction to modify that data item started. This is clearly wrong, and we can see that reverse prcessing gets us the desired result.

The second phase of recovery is to redo the changes done by committed transactons, so that all data items which have been modified by them are restored to the value they had after the *last* of the committed transactions finished. It can be seen that only forward processing of log-records belonging to redo-list transactions can guarantee this.

17.8 Answer: The initial ordering of the disk blocks is: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Assume that the two blocks following block 10 on the disk, are the first two blocks in the list of free blocks.

a. The first 3 **read** steps result in blocks 3, 7, 5 being placed in the buffer.
b. The 4th **read** step requires no disk access.
c. The 5th **read** step requires block 1 to be read. Block 7 is the least recently used block in the buffer, so it is replaced by block 1.
d. The 6th step is to modify block 1. The first free block is removed from the free block list, and the entry 1 in the *current page table* is made to point to it. Block 1 in the buffer is modified. When dirty blocks are flushed back to disk at the time of transaction commit, they should be written to the disk blocks pointed to the updated current page table.
e. The 7th step causes block 10 to be read. Block 5 is overwritten in the buffer since it is the least recently used.
f. In the 8th step, block 3 is replaced by block 5, and then block 5 is modified as in the 6th step.

Therefore the final disk ordering of blocks is: 2, 3, 4, 6, 7, 8, 9, 10, 1, 5. The set of blocks in the buffer are: 5 (modified), 10, 1 (modified). These must be flushed to the respective disk blocks as pointed to by the current page table, before the transaction performs commit processing.

17.11 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.

17.12 Answer:

- 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.

- Let the erroneous transaction be T_e.
  - ☐ Identify the latest checkpoint, say C, in the log before the log record $< T_e, START >$.
  - ☐ Redo all log records starting from the checkpoint C 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.
  - ☐ 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.
- 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.

17.13 Answer: This is implemented by using `mprotect` to initially turn off access to all pages, for the process. When the process tries to access an address in a page, a protection violation occurs. The handler accociated with protection violation then requests a write lock on the page, and after the lock is acquired, it writes the initial contents (before-image) of the page to the log. It then uses `mprotect` to allow write access to the page by the process, and finally allows the process to continue. When the transaction is ready to commit, and before it releases the lock on the page, it writes the contents of the page (after-image) to the log. These before- and after- images can be used for recovery after a crash.

This scheme can be optimized to not write the whole page to log for undo logging, provided the program pins the page in memory.

17.14 Answer: We can maintain the LSNs of such pages in an array in a separate disk page. The LSN entry of a page on the disk is the sequence number of the latest log record reflected on the disk. In the normal case, as the LSN of a page resides in the page itself, the page and its LSN are in consistent state. But in the modified scheme as the LSN of a page resides in a separate page it may not be written to the disk at a time when the actual page is written and thus the two may not be in consistent state.

If a page is written to the disk before its LSN is updated on the disk and the system crashes then, during recovery, the page LSN read from the LSN array from the disk is older than the sequence number of the log record reflected to the disk. Thus some updates on the page will be redone unnecessarily but this is fine as updates are idempotent. But if the page LSN is written to the disk to before the actual page is written and the system crashes then some of the updates to the page may be lost. The sequence number of the log record corresponding to the latest update to the page that made to the disk is older than the page LSN in the LSN array and all updates to the page between the two LSNs are lost.

Thus the LSN of a page should be written to the disk only after the page has been written and; we can ensure this as follows: before writing a page containing the LSN array to the disk, we should flush the corresponding pages to the disk. (We can maintain the page LSN at the time of the last flush of each page in the buffer separately, and avoid flushing pages that have been flushed already.)