Ferdinand Lee
002698658

8.2.4
<START T>
<T, A, 10>
<START U>
<U, B, 20>
<T, C, 30>
<U, D, 40>
<COMMIT U>
<T, E, 50>
<COMMIT T>

Using UNDO LOGGING and last log record to appear on the disk is as follows:
a) <START U>
If the last entry on the log is <START U>, we know that neither U nor T have been committed. Therefore, both transactions U and T must be undone. Scanning the log from the end of the file, we first find that there are no changes for U. Therefore, we can record <ABORT U> in the log. The next item is A. A has not been committed and therefore needs to be undone. We set A to 10 and write it to the disk. Then we write a record <ABORT T> in the log. Finally, the log is flushed.

b) <COMMIT U>
If the last entry on the log is <COMMIT U>, we know that U has been successfully written to the disk. Therefore, we do not need to undo the transactions for U. Scanning the log from the end of the file, we find that T has active transactions but has not been committed. Therefore, we undo the transactions for T. First we set C to 30 and A to 10 and write out the changes to the disk. Then, a record <ABORT T> is written to the log file. Finally, the log is flushed.

c) <T, E, 50>
If the last entry is <T, E, 50>, we can see that U has been successfully committed and written to the disk. However, since we have not seen a commit transaction for T, T has to be undone. Scanning from the end of the log file, we set E to 50, C to 30, and A to 10 and write the changes to the disk. Then we write a <ABORT T> record in the log. Finally, the log is flushed.

d) <COMMIT T>
If the crash occurs after <COMMIT T>, we can see that both transactions U and T have been successfully written to the disk and committed. There are no other active transactions. Therefore, no further changes are necessary.

8.4.2
Since this exercise follows the section in undo/redo logging, I will assume that this is what the author meant for this exercise and not undo logging.
a)
<START T>
<T, A, 10, 11>
<T, B, 20, 21>
<COMMIT T>

Since this is undo/redo logging, the constraints enforced by this type of logging allows the COMMIT log record to precede or follow any of the changes to the database elements on the disk.

Let
A: database element A written to the disk
B: database element B written to the disk
LA: database element LA written to the disk
LB: database element LB written to the disk
COM: COMMIT record and write to disk

LA A LB B COM
LA LB A B COM
LA LB A COM B
LA LB B A COM
LA LB B COM A
LA LB COM A B
LA LB COM B A
LA A LB COM B

If this logging were truly a pure undo logging, only the first two would be allowed.

b)
Let
A: database element A written to the disk
B: database element B written to the disk
C: database element C written to the disk
LA: database element LA written to the disk
LB: database element LB written to the disk
LC: database element LC written to the disk
COM: COMMIT record and write to disk

LA LB LC A B C COM
LA LB LC A B COM C
LA LB LC A C B COM
LA LB LC A C COM B
LA LB LC A COM B C
LA LB LC A COM C B

LA LB LC B C A COM
LA LB LC B C COM A
LA LB LC B A C COM
LA LB LC B A COM C
LA LB LC B COM C A
LA LB LC B COM A C

LA LB LC C B A COM
LA LB LC C B COM A
LA LB LC C A B COM
LA LB LC C A COM B
LA LB LC C COM B A
LA LB LC C COM A B

LA LB LC COM B A C
LA LB LC COM B C A
LA LB LC COM A B C
LA LB LC COM A C B
LA LB LC COM C B A
LA LB LC COM C A B

LA A LB LC B C COM
LA A LB LC B COM C
LA A LB LC C B COM
LA A LB LC C COM B
LA A LB LC COM C B
LA A LB LC COM B C

LA LB A B LC C COM
LA LB A B LC COM C
LA LB B A LC C COM
LA LB B A LC COM C

LA A LB B LC C COM
LA A LB B LC COM C

LA LB A LC B C COM
LA LB A LC B COM C
LA LB A LC C B COM
LA LB A LC C COM B
LA LB A LC COM B C
LA LB A LC COM C B

LA LB B LC A C COM
LA LB B LC A COM C
LA LB B LC C A COM
LA LB B LC C COM A
LA LB B LC COM A C
LA LB B LC COM C A

If this logging were truly a pure undo logging, only those with a COM as the last entry are allowed.

8.4.5
<START S>;
<S, A, 60, 61>;
<COMMIT S>;
<START T>;
<T, A, 61, 62>;
<START U>;
<U, B, 20, 21>;
<T, C, 30, 31>;
<START V>;
<U, D, 40, 41>;
<V, F, 70, 71>;
<COMMIT U>;
<T, E, 50, 51>;
<COMMIT T>;
<V, B, 21, 22>;
<COMMIT V>

Suppose that we begin a nonquiescent checkpoint immediately after one of the following log records has been written in (memory):

For each, tell: (i) At what points could the <END CKPT> record be written, and (ii) For each possible point at which a crash could occur, how far back in the log must we look to find all possible incomplete transactions. Consider both the case that the <END CKPT>; records was or was not written prior to the crash.

(a) <S, A, 60, 61>.

i) For undo/redo logging, the <END CKPT> can be written anywhere after the <START CKPT>. The <END CKPT> record indicates that all the “dirty”, meaning changed in the buffer but not written to the disk, transactions that have been committed before the start of the checkpoint are written to the disk. This action is independent of how the database elements are saved to the disk. Therefore, it can appear anywhere after <START CKPT>.

ii) Regardless of whether <END CKPT> was or was not written prior to the crash, we need to go back to <START S> because there were no committed records prior to <START CKPT>.

(b) <T, A, 61, 62>.

i) For undo/redo logging, the <END CKPT> can be written anywhere after the <START CKPT>. The <END CKPT> record indicates that all the "dirty", meaning changed in the buffer but not written to the disk, transactions that have been committed before the start of the checkpoint are written to the disk. This action is independent of how the database elements are saved to the disk. Therefore, it can appear anywhere after <START CKPT>.

ii) If the crash occurs before the record <END CKPT> is written, we have to go back as far as the start of the previous <START CKPT>. Since there are no other checkpoints in the log file, we have to examine all the records of the log file up to <START S>. If the crash occurs after <END CKPT>, we know that all the transactions that have been committed prior to the start of the checkpoint have been written to the disk. Therefore, we need only to examine the active transactions at the start of the checkpoint. Therefore, we need to look only as far back as <START T>.

(c) <U, B, 20, 21>.

i) For undo/redo logging, the <END CKPT> can be written anywhere after the <START CKPT>. The <END CKPT> record indicates that all the "dirty", meaning changed in the buffer but not written to the disk, transactions that have been committed before the start of the checkpoint are written to the disk. This action is independent of how the database elements are saved to the disk. Therefore, it can appear anywhere after <START CKPT>.

ii) If the crash occurs before the record <END CKPT> is written, we have to go back as far as the start of the previous <START CKPT>. Since there are no other checkpoints in the log file, we have to examine all the records of the log file up to <START S>. If the crash occurs after <END CKPT>, we know that all the transactions that have been committed prior to the start of the checkpoint have been written to the disk. So the transaction for S is complete and written to the disk. Therefore, we need only to examine the active transactions at the start of the checkpoint. Since there are two active transactions T and U, we need to look only as far back as <START T> because T has the earliest start point of the two.

(d) <U, D, 40, 41>.

i) For undo/redo logging, the <END CKPT> can be written anywhere after the <START CKPT>. The <END CKPT> record indicates that all the "dirty", meaning changed in the buffer but not written to the disk, transactions that have been committed before the start of the checkpoint are written to the disk. This action is independent of how the database elements are saved to the disk. Therefore, it can appear anywhere after <START CKPT>.

ii) If the crash occurs before the record <END CKPT> is written, we have to go back as far as the start of the previous <START CKPT>. Since there are no other checkpoints in the log file, we have to examine all the records of the log file up to <START S>. If the crash occurs after <END CKPT>, we know that all the transactions that have been committed prior to the start of the checkpoint have been written to the disk. So the transaction for S is complete and written to the disk. Therefore, we need only to examine the active transactions at the start of the checkpoint. Since there are now three active transactions T, U, and V, we need to look only as far back as <START T> because T has the earliest start point of the three.

(e) <T, E, 50, 51>.

i) For undo/redo logging, the <END CKPT> can be written anywhere after the <START CKPT>. The <END CKPT> record indicates that all the "dirty", meaning changed in the buffer but not written to the disk, transactions that have been committed before the start of the checkpoint are written to the disk. This action is independent of how the database elements are saved to the disk. Therefore, it can appear anywhere after <START CKPT>.

ii) If the crash occurs before the record <END CKPT> is written, we have to go back as far as the start of the previous <START CKPT>. Since there are no other checkpoints in the log file, we have to examine all the records of the log file up to <START S>. If the crash occurs after <END CKPT>, we know that all the transactions that have been committed prior to the start of the checkpoint have been written to the disk. So the transaction for S and U are complete and written to the disk. Therefore, we need only to examine the active transactions at the start of the checkpoint. Since there are two active transactions T and V, we need to look only as far back as <START T> because T has the earliest start point of the three.