UNDO LOG

The key idea of undo log is "In this transaction T I will change values. To be safe, I will save the OLD values to the log on disk. Only after that would I change the old values to new values on disk. Once I'm done I will write <commit T> to the log on disk to indicate that I'm done."

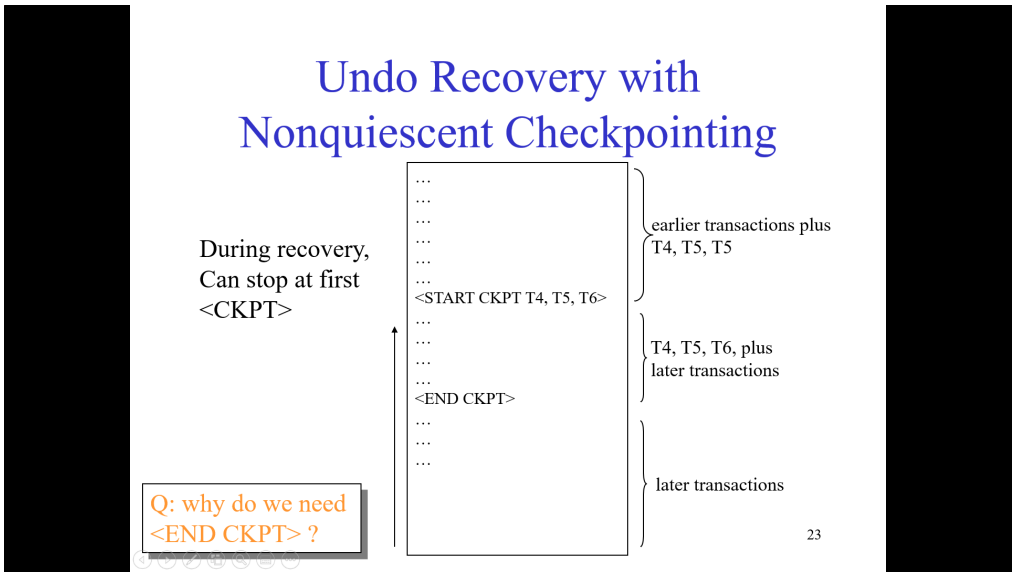
Thus, per the rule of the undo log (as described on lecture slides), if you see <commit T>, then all changes by T must have been on disk.

Put differently, when the system crashes, all transactions that have completed are DONE. Here a transaction is completed if it has been committed, that is, has written <commit T> to the log.

But if a transaction is not yet completed (no <commit T> on log), then we don't know what happens. Part of the changes by the transaction may have made it to disk, and part may not. So we must UNDO the changes that may have made it to disk. Luckily, the OLD values have been saved to the log on disk, so we can use them to undo the changes.

So the recovery rule is simple: read log from the end, if a transaction has committed, then ignore its records. Otherwise if a transaction is not yet committed (you haven't seen the <commit T> record), must undo possible damage: if you see <T,X,v> then reset X to its old value, which is v.

In case of checkpointing. Consider the scenario below:



When we write <start ckpt T4, T5, T6>, we are saying that the ONLY transactions that are active at that point are T4, T5, T6. All other transactions must have committed by then.

When T4, T5, T6 have also committed (that is, writing <commit T> to log), then we write <end ckpt>. It follows that at the point when <end ckpt> is written to log, ALL transactions earlier than T4, T5, T6, PLUS T4, T5, T6 must have committed. So we can ignore them.

The only transactions we can't ignore are those starting after <start ckpt T4, T5, T6>. So it follows that

- when we process the undo log from bottom up, if we see a <end ckpt> record, then we must continue process the log until we see the <start ckpt>. But we don't need to process beyond this record.
- if we don't see any <end ckpt> record, then we must process the log all the way up to where T4, T5, T6 start (whichever of these did not commit). We can't stop at the <start ckpt ...> record.

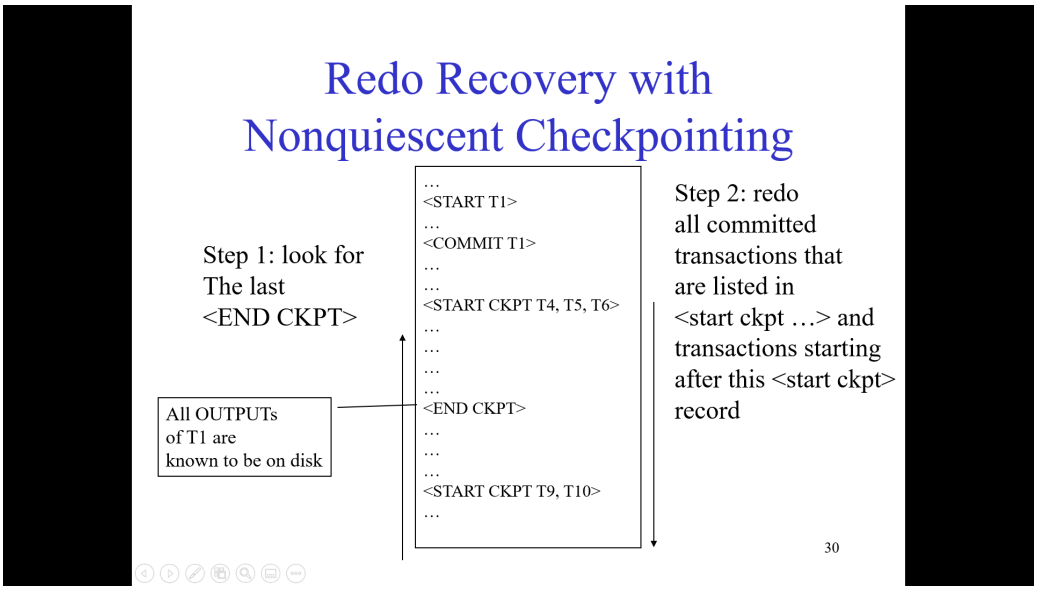
REDO LOG

The basic idea of redo log is "In this transaction T I will change values. So I will write the NEW values and a <commit T> record to the log on disk. Only AFTER that would I start changing the old values to the new values."

Thus, in redo logging, if we see a transaction T in the log and there is no record <commit T>, then none of T's changes has been on disk. (Recall that with this log, we must write a record <commit T> to disk BEFORE we start flushing the changes of this transaction to disk.) So when recovering, we can ignore T.

But if we see <commit T> in the log, then some changes by T may have made it to disk, but some other changes may have not. So when recovering, we need to address this, by redo-ing the entire transaction T. We go through its log records and for each record <T,X,v>, we reset X to the new value v.

In case of checkpointing, consider the example below:



When we write <start ckpt t4, t5, t6>, we are saying t4, t5, t6 are the ONLY active transactions at that point. Then we start flushing to disk all changes made by transactions that have been committed up until that point (of when we write <start ckpt ...>). In this case, suppose the only transaction that has committed is t1 (you can see a record <commit t1> on the picture). Then we are flushing the changes of t1 to disk.

When we are DONE with flushing the changes, we write <end ckpt>. See the picture.

So now when recovering, we look for the last <end ckpt>. If one exists, then we know that all transactions that have committed BEFORE the <start ckpt ...> record have been on disk, and we don't have to worry about them.

But we still have to worry about those transactions that are listed in <start ckpt ...> (which are t4, t5, t6 in this case) AND that have been committed. We have to redo those transactions AND also transactions that start after the <start ckpt ...> record and that have been committed.

To elaborate on the above example, assume the log is:

<start t1>
<t1, x, 5>
<start t4>
<t4, y, 3>
<commit t1>
<start t5>
<t5, x, 2>
<start t6>
<start ckpt t4,t5,t6>
<t4, y, 5>
<start t7>
<end ckpt>
<commit t4>
<t7, z, 4>
<commit t7>

The last <end ckpt> record is the blue record, and the corresponding <start ckpt> record is the red record. The <start ckpt> record lists 3 active transactions: t4, t5, t6. Among these, only t4 has committed. The only transaction that starts AFTER the <start ckpt> record is t7, and it has also committed.

So we must redo the two transactions: t4 and t7, from top down. So we start with <t4,y,3> and set y value to 3, then <t4,y,5> and set y value to 5, then <t7,z,4> and set z value to 4. (We have to process records in orange color in the log.)