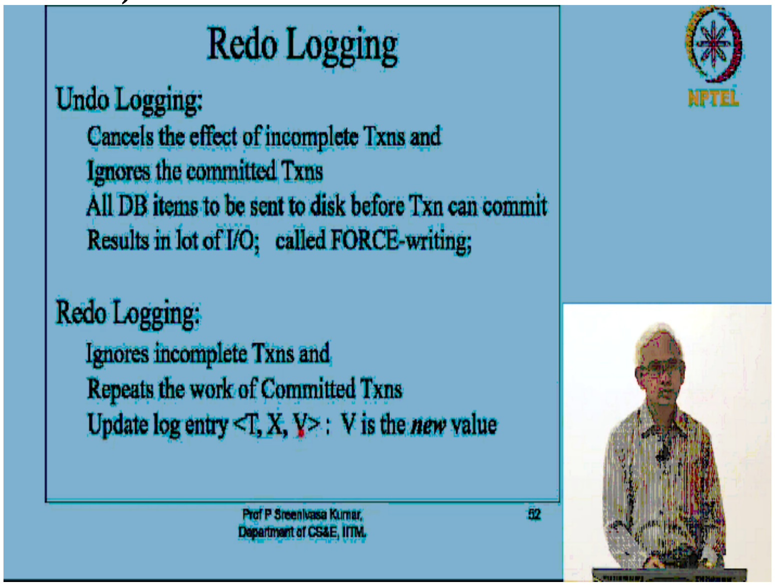
Database Systems
Prof. Dr. Sreenivasa Kumar
Department of Computer Science and Engineering
Indian Institute of Technology – Madras

Lecture - 40
Recovery using Redo and Undo-Redo logging methods

(Refer Slide Time: 00:19)



Redo Logging

Undo Logging:
- Cancels the effect of incomplete Txns and
- Ignores the committed Txns
- All DB items to be sent to disk before Txn can commit
- Results in lot of I/O; called FORCE-writing;

Redo Logging:
- Ignores incomplete Txns and
- Repeats the work of Committed Txns
- Update log entry <T, X, V> : V is the *new* value

Prof P Sreenivasa Kumar, Department of CS&E, IITM. 52

So, let us get started so we have been discussing the various ideas and concepts, the recovery subsystem of the relational database. So, the principle tool that we use log and log is a sequential file where we keep making entries about the transactions that are under progress and we keep certain important information recorded in the log and we use that log to recover from failures.

So, in that context, we looked at what are the various log entries and then we started looking at various specific logging methods and we looked at protocol and undo logging. In this lecture, we will focus on other logging methods, particularly what is called redo logging and also undo redo logging we will see that. So, in undo logging as we have seen in the last lecture, it basically cancels the effect of incomplete transactions.

So keeps track of the old values of the items or the date the transaction is in progress have change in the log entries and making use of these old entries to cancel the effect of incomplete transactions and because of the way it has been made. We do not need to specifically do anything about committed transaction now, but one major issue with this particular way of logging is that the database items.

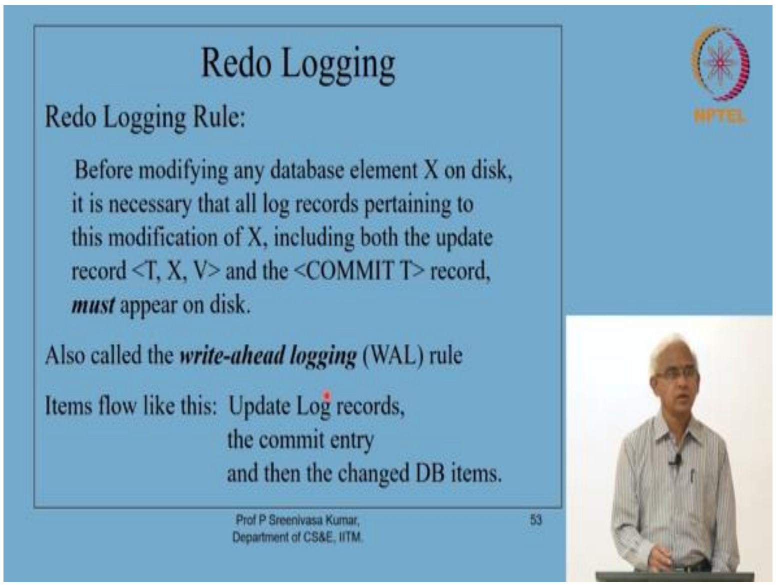
All the database items that the transactions have changed, a particular transaction has changed, have to be return in an urgent manner to the disk before we allow the transaction to commit. So, this is what is called force writing and so we do not have much flexibility in the matter. So, suppose there is a blog that has been actually request is probably useful for many transactions and they are going to be changing that.

So, we will not be able to kind of accumulate the changes made by several transactions and then write this block at a time you know we will not be able to do that. So, whatever may be the changes that we are there now, when this particular transaction which is which has made use of this blocks wants to commit, then we have to send that block back to the disk. Only then we can make a commit, so this kind of results in a lot of I/O.

So is there any other so before we proceed to undo redo log in let us look at the in some sense the inverse of this kind of method called the redo logging, where we ignore the incomplete transactions and then we repeat the work of committed transaction. So, obviously, if you repeat the work of the committed transactions, the log entries that we have to made use of now have to be different.

So, in the log entries now we will keep track of the new value whatever the transaction is changing from database actual changes will keep track of the new value. So, the update logs every all other log entry will remain as they are. The update log entry will now become transaction ID, what is the item and what is a new value?

(Refer Slide Time: 03:53)



Redo Logging

Redo Logging Rule:

Before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X, including both the update record <T, X, V> and the <COMMIT T> record, ***must*** appear on disk.

Also called the ***write-ahead logging*** (WAL) rule

Items flow like this: Update Log records, the commit entry and then the changed DB items.

Prof P Sreenivasa Kumar, Department of CS&E, IITM. 53

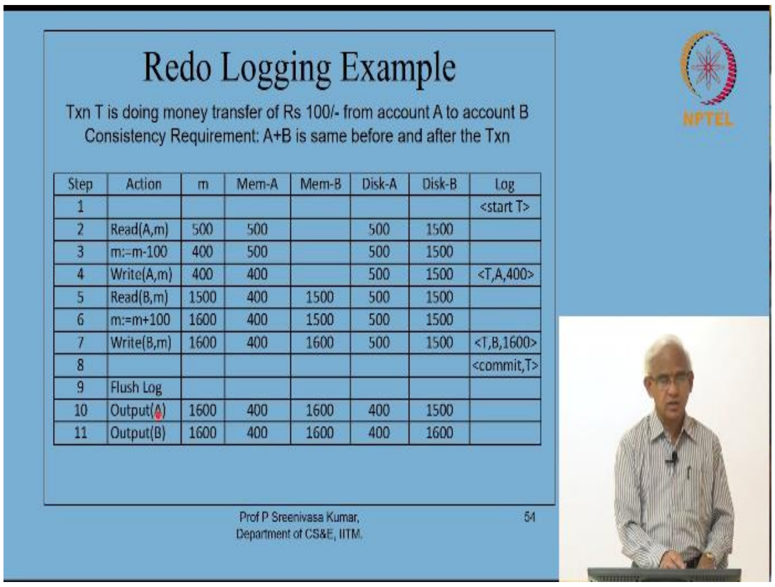
And we will see how exactly are they different method procedures, so the redo logging method is simple, so before this is also called write ahead logging. So, before we write anything on to the database, we must ensure that all the log entries made by that particular transaction are all on the disk and including the COMMIT T entries data like this. So, before modifying any database item X on the disk.

It is necessary that all raw log records pertaining to this modification, including the update records as well as the COMMIT T record must reach this must appear. So, the actual update of the items on the disk can proceed in a leisurely manner afterwards but as soon as possible. So, we have that flexibility now, so this particular principle is what is called write ahead logging rule. So, the items will flow like this, suppose the update log records.

We will go to the disk and then the commit T entry will commit entry will go to the disk and then actual changes to the database on the disk will be carried out in a little early manner. So, this will not you know there is no particular hurry to write all the items to the disk. The transaction is not you know kind of waiting to commit. So, it does the commit entry, so we allow it to go.

The main reason why we can get away with that is that we have a record of what are the changes that particular transaction has done in the form of the update because we have new values and so, in case the transaction itself fails to do the updating of the database. We will be able to do that with the help of the log records, so that is the idea in this.

(Refer Slide Time: 06:00)



Redo Logging Example

Txn T is doing money transfer of Rs 100/- from account A to account B
Consistency Requirement: A+B is same before and after the Txn

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

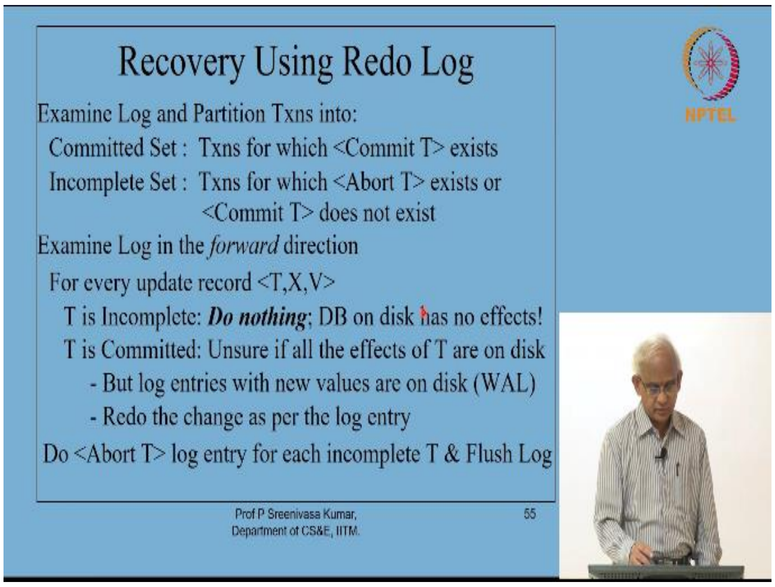
Prof P Sreenivasa Kumar,
Department of CS&E, IITM. 54

So, let us go back to this redo example of 100 rupees being change transfer from account A to account B the same example. So, you are familiar with all these columns now, so, we are using a single variable and the actual method is same. So, we read A into m reduce it by 100 write it read B into m increased by 100 and write it. Now, the only thing here is that we do a commit A and so let us focus on the log now.

The log records will now give the new values, so when write A, m has occurred, what we are keeping track now is the transaction T has changed this DB item A and the new value is 400. So that was the last entry and then transaction in a similar way, transaction T has changed the transaction has changed the item B and it is new value is 1600. So, these are the log entries and as soon as we do a work as soon as a transaction complete it is work.

It will make a commit T entry. So, that means it will request for a commit and it will actually internally there will be a flush log and so, all these entries will be permanently there on the log file including the commit T then the transaction is T into B committed. Now, in allegedly manner, there may be some other interleaved operations here of other transactions but at some point, of time we will do an output A output B where we will send this item A to be disk. Now let us also again look at the various scenarios and then see how exactly we will recover from this.

(Refer Slide Time: 08:03)



Recovery Using Redo Log

Examine Log and Partition Txns into:
  Committed Set : Txns for which <Commit T> exists
  Incomplete Set : Txns for which <Abort T> exists or <Commit T> does not exist
Examine Log in the *forward* direction
  For every update record <T,X,V>
    T is Incomplete: ***Do nothing***; DB on disk has no effects!
    T is Committed: Unsure if all the effects of T are on disk
      - But log entries with new values are on disk (WAL)
      - Redo the change as per the log entry
  Do <Abort T> log entry for each incomplete T & Flush Log

Prof P Sreenivasa Kumar, Department of CS&E, IITM. 55

So the recovery method is like this, we will exam so if a crash occurs, we will the recording module is the first one that gets control of the system, it will open the log file and then figure it out as to what is there in the log and then examine the log partition the transactions into 2 sets committed set and the incomplete set. The committed set is the one with transactions for which the Commit T exists.

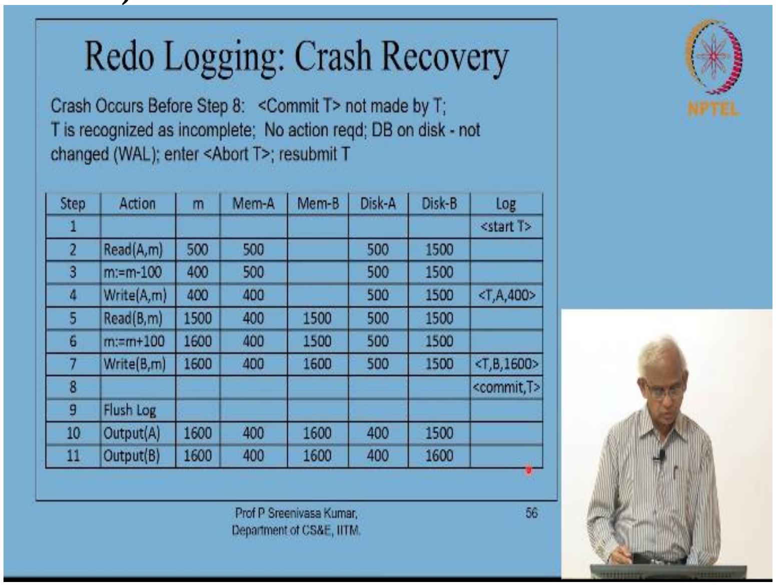
Commit T exists on the log for the incomplete transactions, either Abort T exists or Commit T does not exist. So, we will partition this like that and now we will since we are redoing the work of transactions. We are writing new values. So well have to start from the beginning of the log and then start doing the work of pestering the state of the database. So, we will examine the log in the forward direction remember this was in the reverse direction the case of undo logging.

So, for every update record that we find the transaction T has changed database item and V is new value. So, if T is incomplete, they will not will do anything about that transaction. Let us because if the Commit T record has not read disk and so, we are able to classify it as a committed thing. The direction the transaction would not have attempted to do any changes to the database anyway and so, if it is incomplete transaction will do nothing.

If it is a commit transaction committed set it is committed that means not committed it is there actually in the commit set that means it is commit T record exists in the log. So, then we are kind of unsure if all the effects of this particular transaction have actually reached the disk or not. But fortunately, because of this write-ahead logging principle that we have as part of the redo logging method we have all the log entries which record the new values already on the disk.

So, we will make use of those log entries and then redo all the changes as per the log entries starting from the beginning. And so for every of course, after you do this we will do and we will make an abort T log entry for the incomplete transactions and then to reflect on so that. All this record that we have looked at transaction B and noticed that it is in Abort T transaction is actually available in the log and then of course we will have to resubmit always incomplete transaction, never lose and ensure that they get done.

(Refer Slide Time: 11:06)



Redo Logging: Crash Recovery

Crash Occurs Before Step 8: <Commit T> not made by T;
T is recognized as incomplete; No action reqd; DB on disk - not changed (WAL); enter <Abort T>; resubmit T

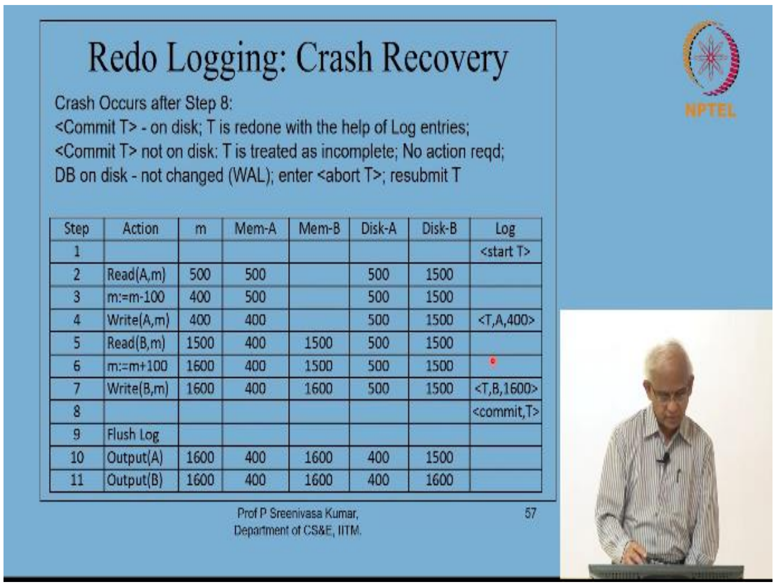
Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

Prof P Sreenivasa Kumar,
Department of CS&E, IITM.

56

So, again in this example, let us look at the various cases the crash occurs before step 8 that means before commit T. So if crash occurs somewhere here, then obviously T is an incomplete data does not matter commit T. So no action is actually required and we basically resubmit this transaction.

(Refer Slide Time: 11:45)



Redo Logging: Crash Recovery

Crash Occurs after Step 8:
<Commit T> - on disk; T is redone with the help of Log entries;
<Commit T> not on disk: T is treated as incomplete; No action reqd;
DB on disk - not changed (WAL); enter <abort T>; resubmit T

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

Prof P Sreenivasa Kumar,
Department of CS&E, IITM.

57

Now, let me change slide so, crash occurs after the step 8, somewhere here crash occurs. So again, there will be 2 cases here so whether the commit T record itself as a change has reached the less or not. That will basically decide the fate of the transaction. The commit T record has reached the disk then we will leave it as committed and then we will redo the work. So, if the commit remember one thing that if commit T has reached the disk.

All these previous entries would have also reached the disk because the log is always written in a sequential manner, you do not write only this entry into the log, you always write all the previous items and the log. So, the moment this is there we know that all these are also there and so we can actually redo the work of this particular topic. So, there are 2 cases we will consider here, the commit T record has actually reached disk and so, we will classify T as a committed transaction.

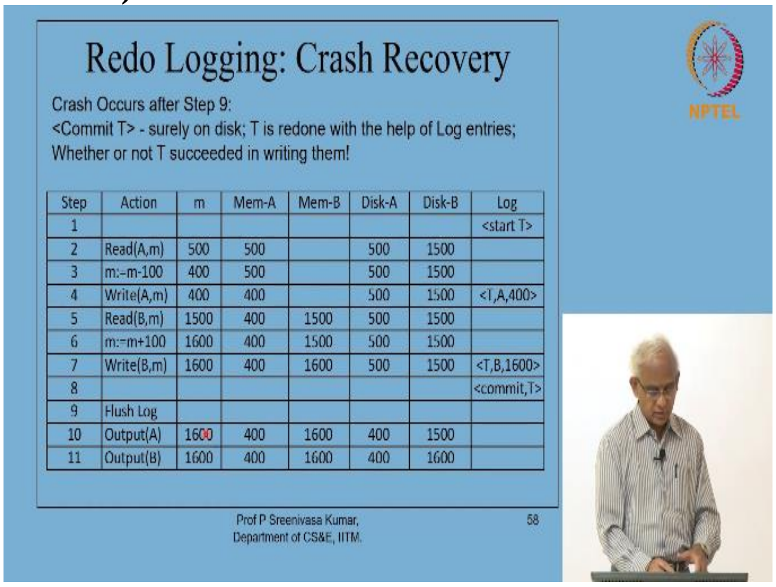
Completed transactions and with the help of these log entries, we will redo the work and the other case where you know the crash occurred but then the flush log itself even before the flush log itself the crash occurred, so the committee actually has not reached disk. So, this is a real unfortunate situation because this flow has actually in fact finished all the work and but unable to manage to get this commit T into this transaction into this disk.

So, but we do not know the situation what exactly has happened we will not know. So what we will do our uniform policy here is that we will not take the risk. We will say that this particular transaction is come again because we really do not know when exactly the crash occurred. So the DB disk T is treated as incomplete and no action is required and we will actually resubmit T, is that clear?

Because see if commit T has not reached disk then we are very sure that see the issue of commit T reaching the disk has to be careful and if the thing has reached disk, then maybe actually you know you will kind of system has you know agreed that the transaction has committed and so we are under obligation to redo it is log but if commit T is not reaching the disk, then we are actually not the system is not obliged to do any changes.

We are also not sure what exactly has happened because some of these log entries might just not be available. We do not know whether we have the complete set of log entries or not we do not know, the commit T not exist. So, we cannot really take any of this I will say that we will resubmit T. So that is the policy of redo logging.

(Refer Slide Time: 15:32)



Redo Logging: Crash Recovery

Crash Occurs after Step 9:
<Commit T> - surely on disk; T is redone with the help of Log entries;
Whether or not T succeeded in writing them!

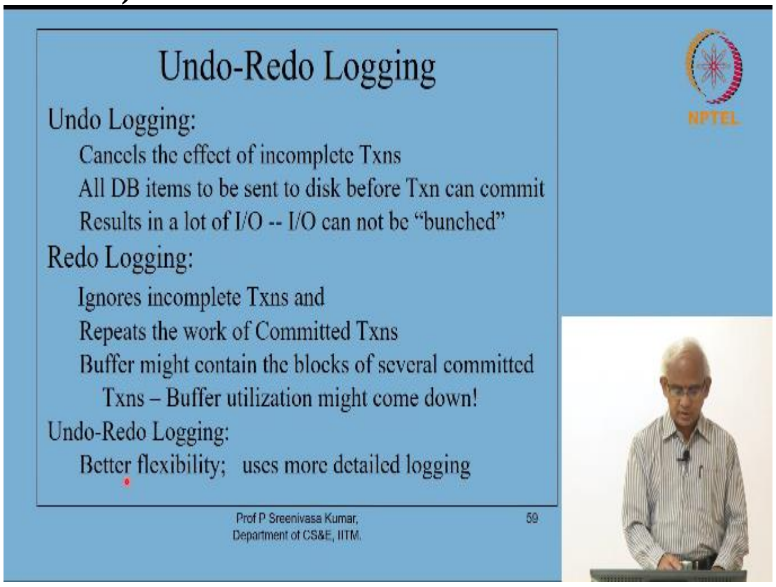
Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:-m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1600>
8							<commit,T>
9	Flush Log						
10	Output(A)	1600	400	1600	400	1500	
11	Output(B)	1600	400	1600	400	1600	

Prof P Sreenivasa Kumar,
Department of CS&E, IITM.
58

Of course if the only crash occurs after Step 9 then we are very sure commit T, surely has reached the disk and so we will simply redo all the work because we have already committed that it is done. So, we have to do the system is our obligation to do the work of the transaction, in this case. We actually will ignore whether it has succeeded to do one output or other output or both of them or something like that. We will simply ignore all those possibilities.

Because this writing of the new values on to be again new values or old values, does not matter if we will do it again. We will do some additional work, which might involve rewriting the same methods but it is safe. So, we will do so that is what the redo logging method is?

(Refer Slide Time: 16:35)



Undo-Redo Logging

Undo Logging:
- Cancels the effect of incomplete Txns
- All DB items to be sent to disk before Txn can commit
- Results in a lot of I/O -- I/O can not be "bunched"

Redo Logging:
- Ignores incomplete Txns and
- Repeats the work of Committed Txns
- Buffer might contain the blocks of several committed Txns – Buffer utilization might come down!

Undo-Redo Logging:
- Better flexibility; uses more detailed logging

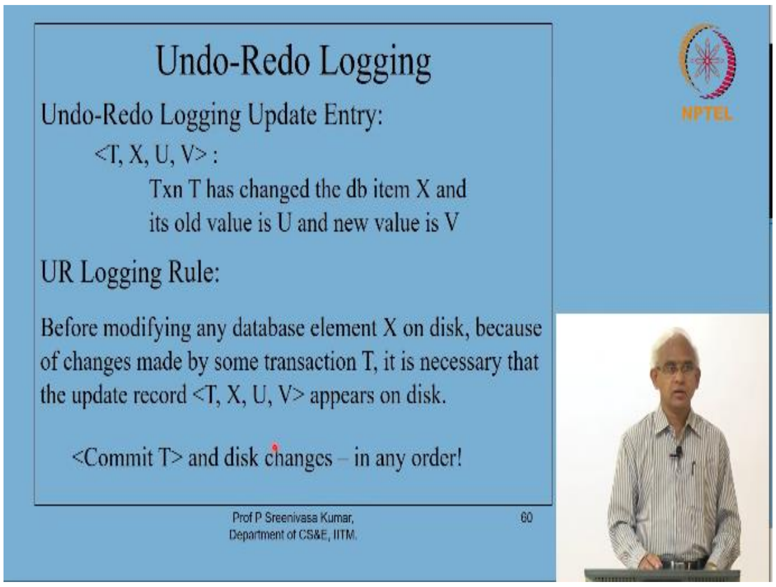
Prof P Sreenivasa Kumar, Department of CS&E, IITM. 59

So, let us now compare these 2 things so the undo logging cancels the effect of incomplete transactions and all the DB items are sent to the disk before the transaction can commit and it results in a lot of I/O and I/O cannot be actually bunched. Bunched means actually, I cannot wait for several transactions do certain changes to a block before I actually shift to the disk. I have to an urgent manner shift the block to the disk.

Whenever a particular whenever a transaction that has that that block commits and redo logging that stress is not there but then only thing is that since we are relaxed about one to you know shift the blocks of the blocks to the disk. It is possible that lots of these blocks are hanging around in the memory and so the buffer utilization actually might come down to when you actually want to bring in a new block from the disk to the buffer.

When you find that there may be space because all these blocks which have been touched by this you know running transactions which have committed have to be there, you know in the memory before because we know that we are going to write them to the disk at some point. So, the there is interesting method which combines these kinds of principles. So this is called undo redo logging where it provides better flexibility. But the expenses that we have to do a bit more detailed logging where we actually have to keep track of both the old values as well as the new values.

(Refer Slide Time: 18:30)



Undo-Redo Logging

Undo-Redo Logging Update Entry:

<T, X, U, V> :

Txn T has changed the db item X and its old value is U and new value is V

UR Logging Rule:

Before modifying any database element X on disk, because of changes made by some transaction T, it is necessary that the update record <T, X, U, V> appears on disk.

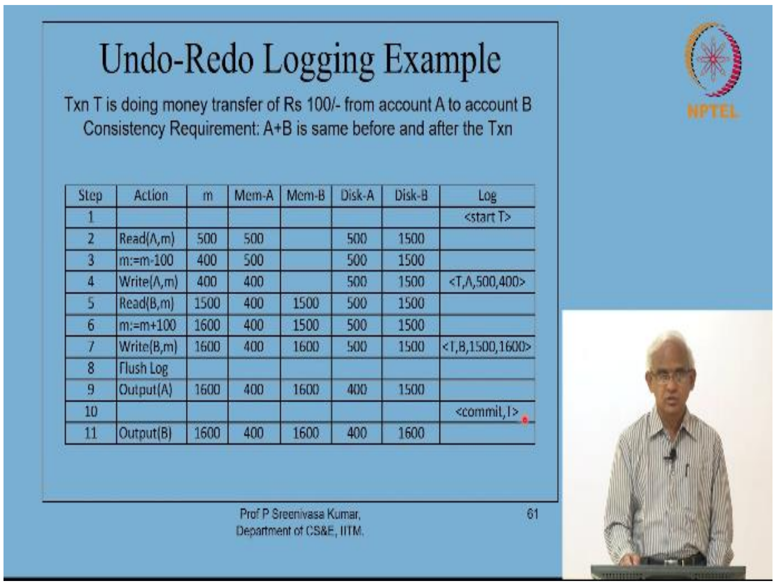
<Commit T> and disk changes – in any order!

Prof P Sreenivasa Kumar, Department of CS&E, IITM. 60

So, in the Undo Redo Logging, the log of the update entry will be actually the transaction ID, the DB item X, what is it is old value and what is new value. So that way we have the flexibility in case we want to roll back a transaction we know the old values. If you want to redo the reduction, we know the new values. So that way we have greater flexibility and so the only principle that we have to follow of course is that these log entries are very important before we do any changes to the database on the disk and sure the log entries reached the disk.

So, before modifying any db item X on the disk, because of changes made by some transaction T, it is necessary that the update because T X U V appears on the disk and the commit on disks changes actually can happen in order suppose commit will happen only after all the operations of a particular transaction has been actually completed and the disk changes can be actually interspersed with commit.

(Refer Side Time: 20:03)



Undo-Redo Logging Example

Txn T is doing money transfer of Rs 100/- from account A to account B
Consistency Requirement: A+B is same before and after the Txn

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,500,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1500,1600>
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10							<commit,T>
11	Output(B)	1600	400	1600	400	1600	

Prof P Sreenivasa Kumar,
Department of CS&E, IITM.

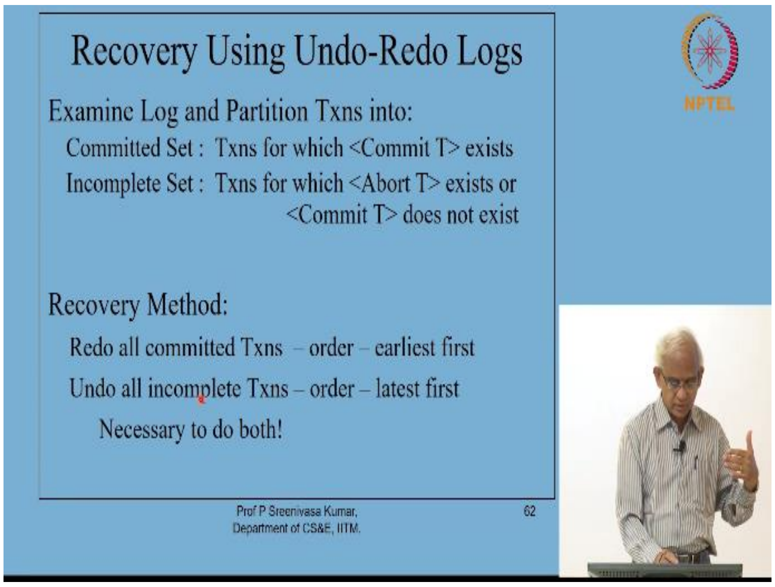
61

So, let us look at the same example where we are now adopting undo redo logging method. So look at the changes that are there in the log entries so when write A, m happens we know that so the log entry to be made is that transaction T has updated this item A, and it is old value is 500 new value is 400 like that we make a detail logging. Notice that these log entries can actually be fairly big because we said what is the lockable item? A lockable item is a block.

So, db item is a block, so they can actually be pretty big logging this even though here technically I am just showing the crucial values here but these data db items are put in the log. In a similar way when transaction write B we will put in the 1500, 1600 and the old value and new value. Now we will do a flush log to ensure as soon as the transaction completes it is work with a flush log because we now are ready to update the database with the changes database on the disk of changes.

So before we do that, we have to make sure that all these log entries have reached the disk. So we do a flush log and then we will actually do so of course it is an advantage for the transaction to kind of ensure that the commit T record request as soon as possible. But it is kind of left to the scheduler to actually look at this possibility. So it might actually change it might actually transfer some disk block to the memory. Even though sorry memory blocks to the disk even though the transaction itself has not made the commit T because it knows that if this is necessary, I can undo the transaction.

(Refer Slide Time: 22:40)



Recovery Using Undo-Redo Logs

Examine Log and Partition Txns into:

Committed Set : Txns for which <Commit T> exists

Incomplete Set : Txns for which <Abort T> exists or <Commit T> does not exist

Recovery Method:

Redo all committed Txns – order – earliest first

Undo all incomplete Txns – order – latest first

Necessary to do both!

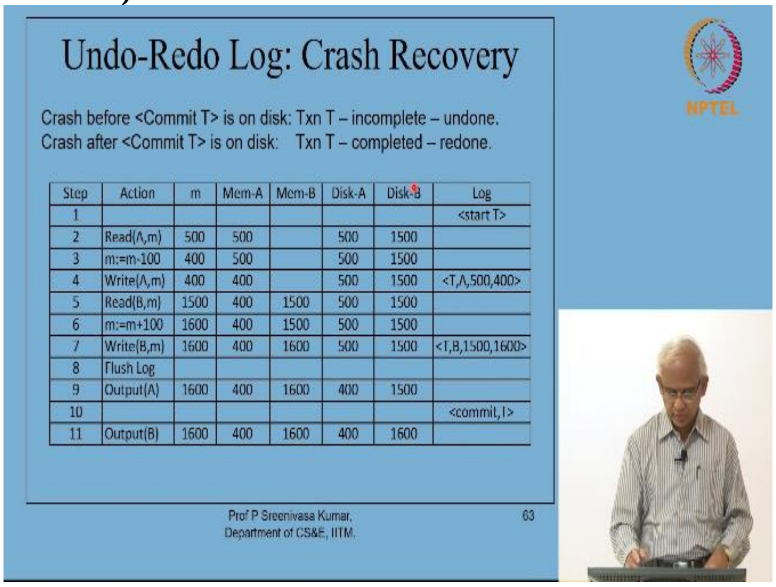
Prof P Sreenivasa Kumar, Department of CS&E, IITM.

62

So again, the principle recovery method is like this again, we are going to after the system wakes up, we will again exam we will examine the log and partition the transactions as you should as we have done earlier whether so everything depends on the commit T exists on the log or not. On the recovery method is redo all the committed transactions in the order earliest first, in the log. So for every transaction, we can actually look at all it is operations in either direction, we can have the entire log with us.

So for the committed transactions, we do a redo step in the forward direction and for all incomplete transactions. We will do a undo in the latest first, and it is of course necessary to do both of them.

(Refer Slide Time: 23:44)



Undo-Redo Log: Crash Recovery

Crash before <Commit T> is on disk: Txn T – incomplete – undone.

Crash after <Commit T> is on disk: Txn T – completed – redone.

Step	Action	m	Mem-A	Mem-B	Disk-A	Disk-B	Log
1							<start T>
2	Read(A,m)	500	500		500	1500	
3	m:=m-100	400	500		500	1500	
4	Write(A,m)	400	400		500	1500	<T,A,500,400>
5	Read(B,m)	1500	400	1500	500	1500	
6	m:=m+100	1600	400	1500	500	1500	
7	Write(B,m)	1600	400	1600	500	1500	<T,B,1500,1600>
8	Flush Log						
9	Output(A)	1600	400	1600	400	1500	
10							<commit,T>
11	Output(B)	1600	400	1600	400	1600	

Prof P Sreenivasa Kumar, Department of CS&E, IITM.

63

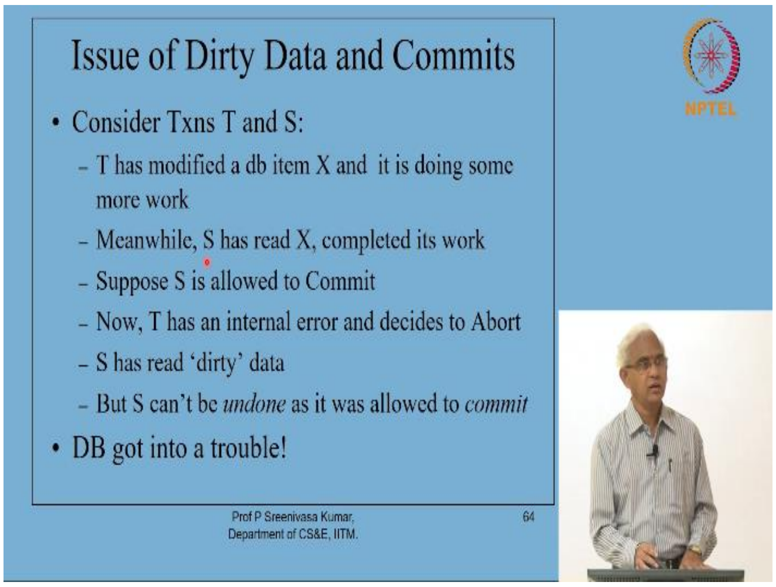
So, let us look at are there any questions now? Let us look at this. The various scenarios the only thing of course, is, so even the crucial thing is this commit T there on the disk or not?

That is the crucial thing as in the predictions. So, if the crash occurs before commit T is on the disk that means some ways, we are in the crash occurs then we are going to deem the transaction T as incomplete and it will be undone even though it has actually completed.

It is work on transferring some of the changes today that disk, so it has done and output A. So some changes have actually reached disk. But then we know that the transaction B is incomplete and since we have the old values with us, we will do an undo and restore the old values. But suppose the crash occurs and then the commit T record is actually on the disk.
So, then we treat this as completed transaction again notice one thing this is not just with respect to this particular transaction.

Remember that transaction operations of several transactions are getting interleaved and they will actually be happening in an interleaved blocking as per some transaction schedule which is following the two phase locking protocol. We have to combine these 2 things; you know we are right now talking from times up only error recovery. But we have already discussed the issues with respect to conflict serializability. So, conflict serializable schedules are actually happening in the system because all the transactions are off or following through this locking protocol and so interleaving is happening.

So this though I am showing only the operations of one particular transaction in reality there are actually operations of several transactions in flow. And so there is a bunch of these transactions which are you know incomplete and a bunch of them just completed and so we have to actually do redone as well as undone operations.
(Refer Slide Time: 26:44)


Issue of Dirty Data and Commits

- Consider Txns T and S:
  - T has modified a db item X and it is doing some more work
  - Meanwhile, S has read X, completed its work
  - Suppose S is allowed to Commit
  - Now, T has an internal error and decides to Abort
  - S has read 'dirty' data
  - But S can't be *undone* as it was allowed to *commit*
- DB got into a trouble!

Prof P Sreenivasa Kumar, Department of CS&E, IITM. 64

So let us come to another interesting issue now look at this scenario. This brings it actually ever absolutely new kind of requirement. So let us look at this again revisit the issue of this dirty data and commits. So suppose look at this scenario consider some transactions T and S. So T has modified a db item X and it is doing some more work. So, it is kind of released the lock on item X, it released so it does not require any more items to be locked.

So, following the two phase locking principles on release the lock on db item and so, it was available for somebody else. So meanwhile S has read and completed it is work. Now, the order among commits we have not actually discussed anywhere. Suppose S is allowed to commit this is a classic case of the dirty data problem. Suppose S is allowed to commit by our transaction system.

And after that, the T which was released the lock on X and proceeding with some work has found out that it does not read any more items to be read but it has formed some internal error and basically decided to abort; now S has read dirty data. So it has read some incomplete it has read some value that is not supposed to be there because this change that was done by S on this item X is now going to be rolled back it will be random.

Because the transaction itself requests for abort so in case either when a crash occurs or actually when abort happen, you know the system is obliged to run this failure recovery methods even for transaction abort methods, not only for crashes. So internally abort the crash comes, then it will take this you know policy of failure recovery and then actually try to undo the operations of that particular transaction for that particular transaction to undo and redo roll back.

But then we have allowed us to go ahead and commit. So, this should never be allowed actually in this because this basically goes against the principle of what is called committing at all. So, if you allowed it to come in, then we are giving a commitment saying that all the work that is accurate done by S is you know is valid and we are going to keep it permanent record of that.

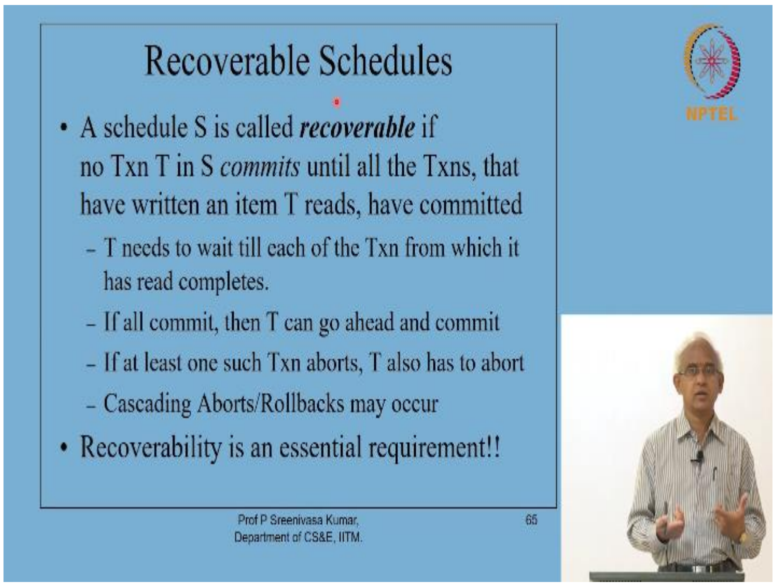
But unfortunately, S the transaction has read some dirty data and it is actually made some mistakes and we cannot do an undo of S because the very meaning of commit is that we are giving a commitment saying that all it is in the record. So we can roll back that particular remember to look at the sequence. So the T has modified item and is actually still learning S has read this item that has changed by T and requested for a commit.

And the system allowed it and later the transaction T has some internal error and request for an abort and the system allowed. If this happens if this sequence of things happen, then we are actually contradicting ourselves because we will not our meaning of commit itself is not proper. So because the ideal thing for us to do is that if T has requested for a rollback. So we should have probably made note saying that this particular transaction S has read some item that has been changed by T make a note of that and rollback that also.

If T is getting rolled back, we should make a note somewhere that S has read some item that has been changed by T and so we should roll back that also. We should stop it and roll it back like that man there is some problem, your transaction you cannot proceed. We have to undo that is the proper way of handling this kind of situation. So even though we did not discuss this kind of interaction so far.

So now we are able to see that if this kind of issue of dirty data arises, then we really have to be careful and we cannot you know independently treats these commits and aborts of transactions in an independent manner. So the DB will get into a trouble.

(Refer Slide Time: 32:32)



Recoverable Schedules

- A schedule S is called ***recoverable*** if no Txn T in S *commits* until all the Txns, that have written an item T reads, have committed
  - T needs to wait till each of the Txn from which it has read completes.
  - If all commit, then T can go ahead and commit
  - If at least one such Txn aborts, T also has to abort
  - Cascading Aborts/Rollbacks may occur
- Recoverability is an essential requirement!!

Prof P Sreenivasa Kumar, Department of CS&E, IITM. 65

So, this particular issue is what is called recoverability we have lost the ability of recovering itself. So, there is a very fundamental requirement. So, this issue has been termed as what is called recoverability recoverable schedules. So let me introduce so serializable schedules we have introduced earlier. Now, let us introduce recoverable schedules are those schedules in which the transaction T commits.

So, no transaction T in the schedule commits until all the transactions that have written an item that this particular transaction is has read have committed. So if I am reading if I am a transaction and I have read an item that has been raised by some running transaction active transaction which has not yet committed, then I should not go on commit because I do not know what is there.

That other fellow is position is whether he will decide to commit or whether you decide to abort I really do not know. So, if I am an active transaction from an active transaction. I have I am doing my work. I have acquired the locks that are raised by other people and done the work. But then I should know of course, there may be no particular way of whether I knowing it but let us see how that can be handled.

But basically, I should be aware that some transaction might have actually changed the item which I have read and so that transaction whether it completes or not, I do not know at this point of time. So if I request for a commit, and actually the system grants that commit, then we are in trouble and so what the system has to do is keep the watch now, of always running transactions and this ensures that this kind of a principle is enforce that.

If transactions request for commits we should ensure that whatever the transaction item that it has read from other transactions you should works that unless those transactions commit it should not allow to go ahead and commit, is that clear? For an active transaction must somehow make a note as to what are the items it has read from other running transactions and we must wait for those other transactions to either commit or abort.

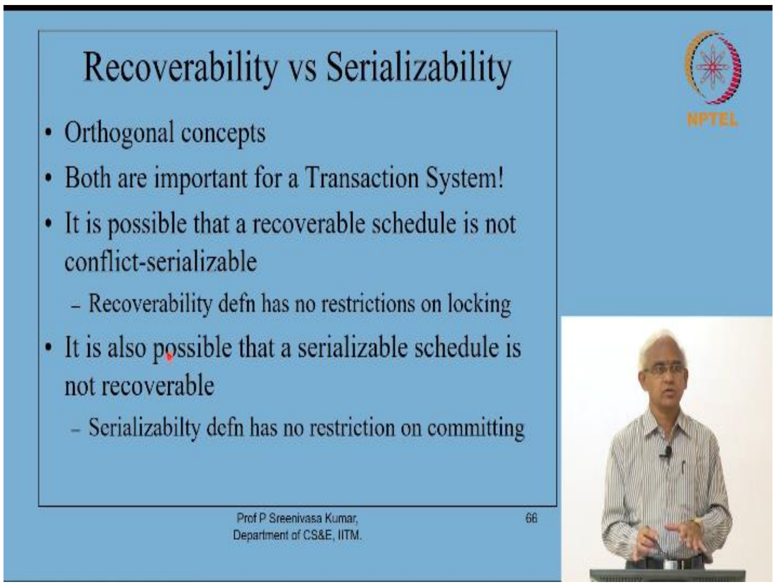
Whatever it is, if all of them commit then I can allow this particular transaction to commit and if one of them aborts. I should abort this fellows that is read some item which has been changed by somebody and that fellow is aborted to abort the fellows. So T needs to wait till each of the transaction promotes it has read completes if all of them commit, then I can go ahead and commit. If one of them does not commit and aborts.

Then this transaction T also has to abort so in such situations what are called cascading aborts work happened cascading aborts, we call them cascading aborts of cascading roll back so because the rollback of one transaction some other transaction will have to be rolled back and because this transaction is getting rollback maybe some other transaction has read some item that has been changed by this little back to the also has to be rolled back.

And everybody has to be a lot of sequence of transactions may have been rolled back that is called a cascading roll back. But recoverability is really an essential feature because to give proper meaning to commit and aborts recoverability is an essential requirements. So, this kind of combines both these issues of interleaving and then the order among commits exactly. So, this is one basic requirement for schedules to satisfy, it turns out that these 2 things the serializability requirement and the recoverability requirement.

This happened to be orthogonal actually we can see that so that means it is possible that for some schedule is actually serializable but not recoverable. Whereas it is recoverable but not serialized.

(Refer Slide Time: 37:51)



Recoverability vs Serializability

- Orthogonal concepts
- Both are important for a Transaction System!
- It is possible that a recoverable schedule is not conflict-serializable
  - Recoverability defn has no restrictions on locking
- It is also possible that a serializable schedule is not recoverable
  - Serializabilty defn has no restriction on committing

Prof P Sreenivasa Kumar,
Department of CS&E, IITM.

66

Basically, that you can see because we do not so recoverability versus serializability these are turning out to be orthogonal concepts. So, it is possible that a recoverable schedule is not conflict serializable because recoverability actually does not put any restrictions on locking in 2 phase. We do not we all we are saying is this who has read from what item and then commit should be in a particular order.

So, it is bothered about commits whereas serializable schedules may not be recoverable because serializability the definition, we can see that it only bothers about what the conflicting pairs of operations and are the same in the same order as any other or some serious schedule that is what serializable schedule So, it does not put any restrictions about committing. So these 2 happen to be orthogonal concepts but we in fact, we want both of them to be satisfied for practical, for operationally correct schedule.

So, schedule asked me both conflictable as well as recoverable for us to be useful whereas for it to be useful for us. So we will see how exactly we can achieve this in the next class actually I think we made consider progress. So the next class will be a better job.