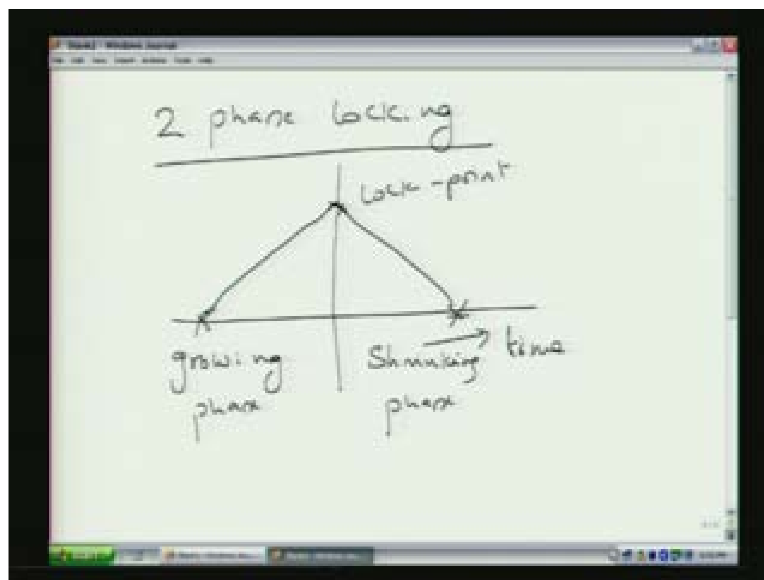
Database Management System
Prof. D. Janakiram
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. 26

Concurrency control for Distributed Transactions

In the earlier lecture, we just looked at how to face commit and three phase commit protocols work what we are going to look at in the lecturer is the concurrency control protocols for distributed system. How this concurrency protocol can be integrated with the commit protocol. In the first few minutes we are going to look at is how the two phase locking protocol can be integrated with the two phase commit protocol for the distributed system. Now what we have is basically in the two phase locking, there are two phases of execution which we have seen earlier.

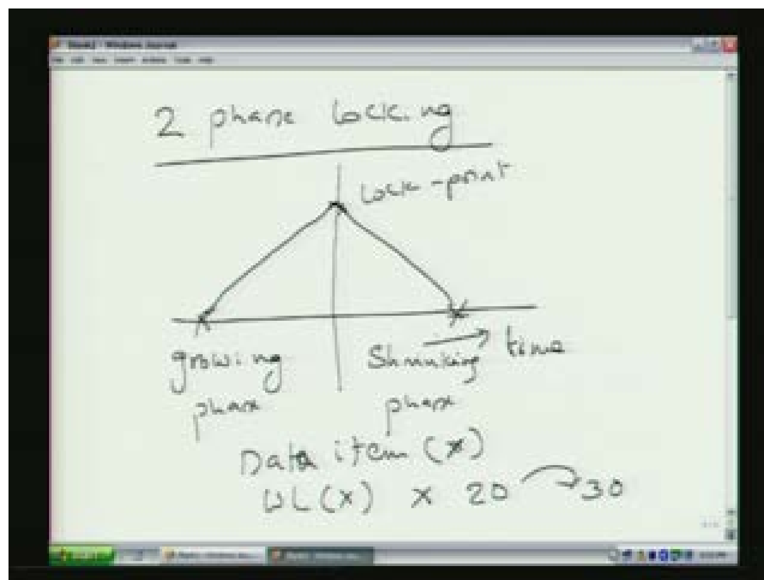
There is basically a growing phase in which the transactions acquires the lock and then there is an actual point were they start releasing the locks and the rule here is that in terms of time, this is the time axis. Now in terms of the time axis, what you are looking at here is a particular phase were this is basically called the growing phase where the transaction is acquiring locks and this is the shrinking phase were it is releasing the locks. Now one of the condition is this is basically called the lock point. So if you have actually released one lock, you cannot ask more locks that is how basically ensures that transaction is always executing in a serializable order. Now, this is basically transaction execution point. For example: this is the start point and this is the end point. Now, if you release a lock for a transaction, then somebody else in between can read the values of this data item that we have actually modified.

(Refer slide time: 3:30)



For example: Assume that even before you have finished you released a lock on a data item which actually means that somebody else can now read the value of that data item. Assume that there is a data item x and now you actually have acquired a write lock on x during the growing phase and then release this lock during the shrinking phase. Now it is possible that when you have actually modified the value of x. Let us say x is now modified from earlier value is twenty, now it got modified itself to thirty. Now during this phase when you release the lock, somebody can acquire this lock, which means that they can read the value of x which you have modified before you have actually committed which means that if you now want to abort, then the transactions which read this value earlier all have to abort.

(Refer Slide Time: 4:32)

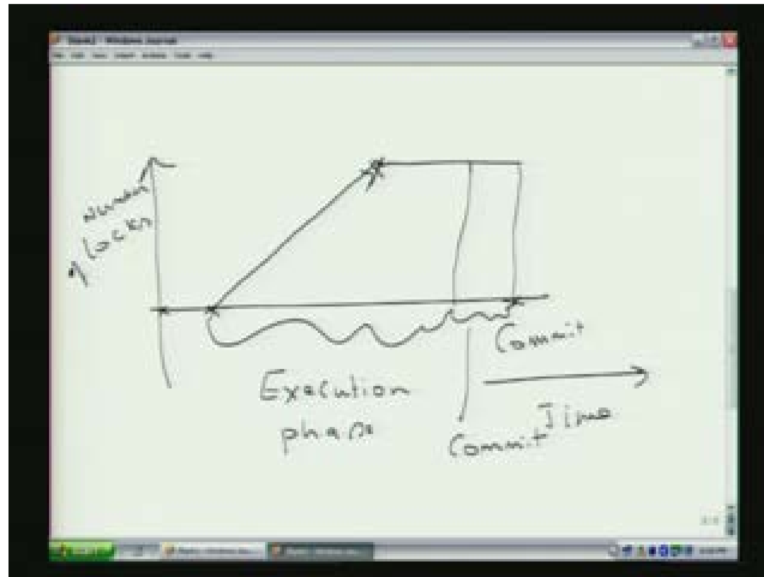


This will result in cascading aborts. If you allow transactions to read the values other transaction to read the values of the data item before the transaction is actually committed. So in general, this execution of the two phase locking is not correct because you cannot release the locks before you have actually committed. So what modification that we are going to get when you integrate with two phase locking commit is you will basically acquire all the logs you will come to the commit point but your lock point and you are going to keep this locks till the end of this transaction, execution, plus the commit phase. Only after the commit phase, after you have committed that this is the point were you have actually commit point your going to release your locks in terms of the time scale.

This is the number of locks that you have. So the number of locks is actually increasing with time. The transactions actually started at some point here and it is started executing it acquired all the locks it required the point actually where it reached it required all the logs, then it continued execution, somewhere here it actually finished execution that means this is the execution phase. Now after the execution is over the commit protocol is going to be started now because then it has to decide to commit, then this is the zone

which the commit protocol gets executed and after the commit protocol is executed, the transaction actually finish execution, then the locks are released at the same time.

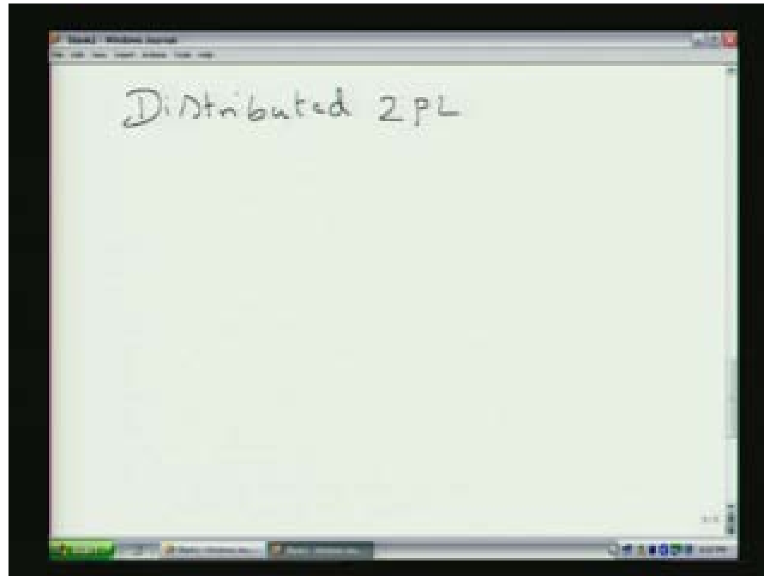
(Refer Slide Time: 6:46)



If you release the locks earlier, this will result in the isolation property being sacrificed and other transaction able to read the values of still uncommitted, modified value of this transaction. So you are not going to release the locks as shown in the earlier diagram but this integrates both the locking and the commit protocols together and after execution is over, you actually run the commit protocol and then release your locks. This is how the two phase locking is integrated with the commit protocol.

Now if you want to see in the distributed setting, how the distributed two phase locking will work. It is actually a simple extension of the two phase locking, because what we now going to see is distributed two PL plus its integration with the two phase commit.

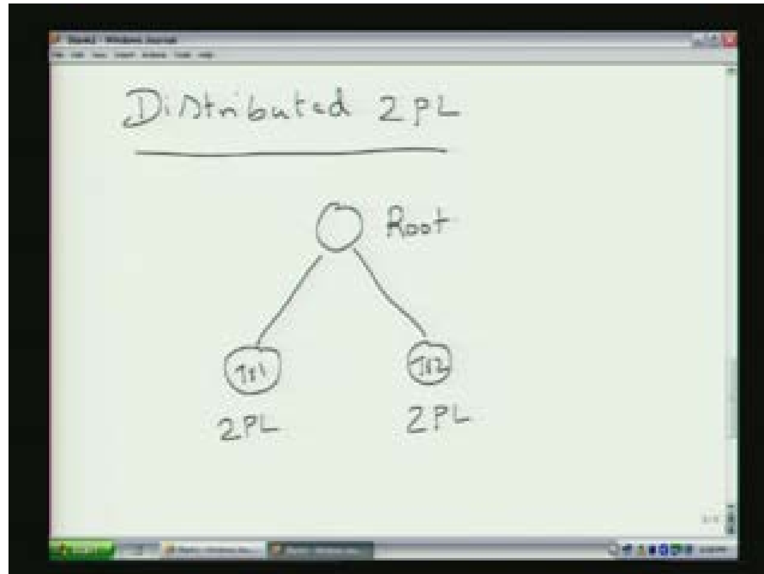
(Refer Slide Time: 7:46)



Two phase commit is for distributed transactions, because commit protocol makes sense only in the case when you have where your transaction is executed. Otherwise, there is no question of involving two phase commit there because you have actually finding out the participants can all commit their transaction. There is only single node that nodes knows know already reaches the commit phase all that it has to now decide is whether its committing the transaction or not. Now, in the case of distributed two PL how you are going to execute the transactions is you basically have a coordinator which we have shown earlier.

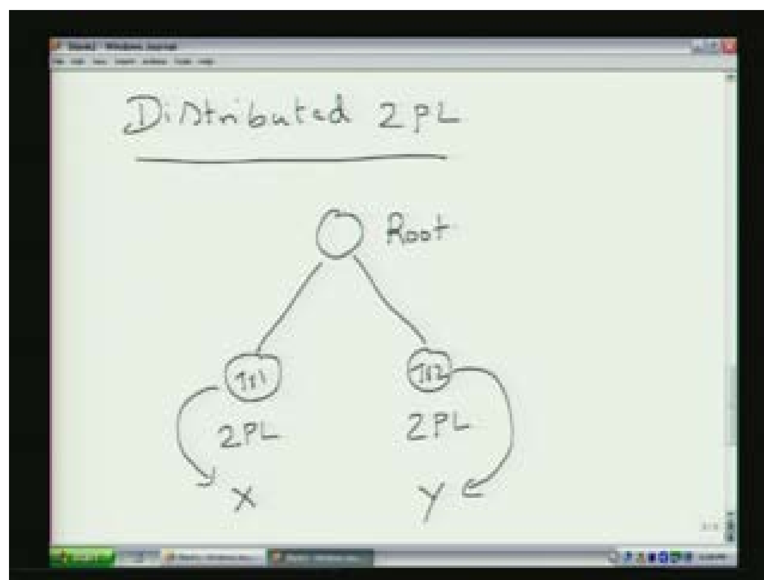
This is basically the route transaction and the route transactions have actually spawn sub transactions which means you actually have the Tr 1 is here and there is a Tr 2 here. Now the Tr 1 actually executes a two PL which actually means that it acquires all the locks that it actually needs and this other transaction also acquires all the locks it needs in terms of the two PL.

(Refer Slide Time: 9:10)



Now each one of them when they reach the commit phase have to decide whether they have are committing or not committing. When the actual commit message has been taken by the coordinator that is the time they will commit and release their locks. Imagine that there is actually a data item X on which this transaction is operating, T1 is operating. There is a data item Y on which the transaction two is operating. Now T 1 will acquire locks on x, T 2 will acquire locks on Y. They finish their part of the execution.

(Refer Slide Time: 9:56)

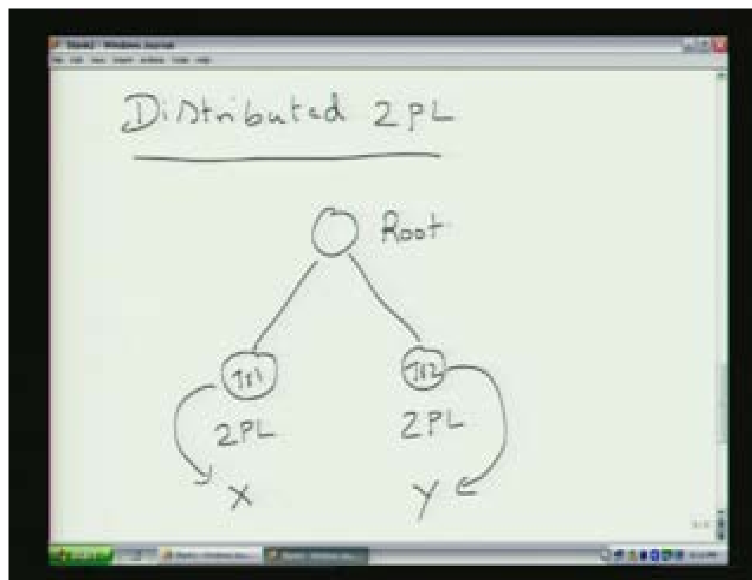


Now at the end of the execution, each one is going to decide whether it is going to commit its part of the transaction that is were two phase commit will start know

execution out two phase commit protocol is run you figure out whether both of these transaction are willing to commit. Now based on that, based on the commit protocol outcome, they will commit the transaction and then release the locks. Now the important thing is what we discuss is the earlier lecturer is what happens when failure occurs then, what is the problem of blocking? Now you assume that T 2 gets blocked. Transaction 2 get blocked which actually means that is not going to release the lock is because it acquired. Since such time it actually made a decision on committing or abort.

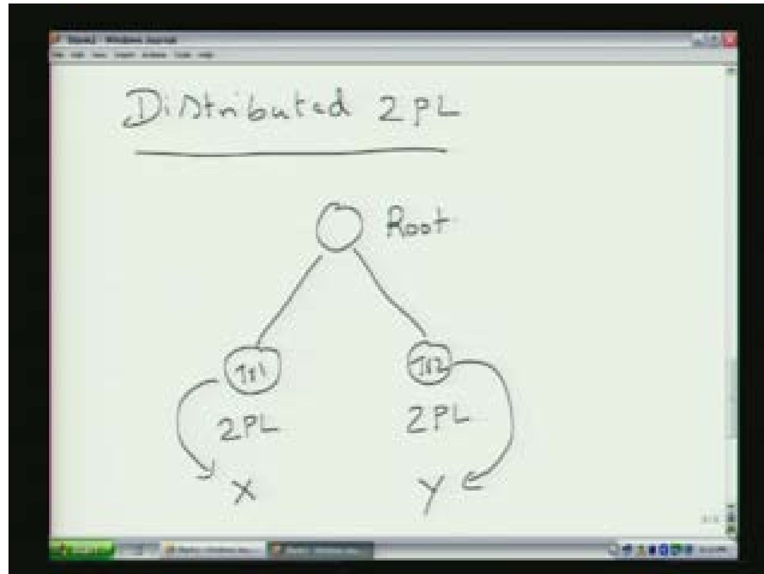
The locks are not released till the commit protocol has actually finished which means that the other transactions which try to acquire locks now on these data items will be blocked from acquiring the locks. So basically blocking in that sense is not good, for the simple reason that the resources are blocked. In this case, data items are blocked from access by other nodes. What i am going to show you is a slight extension of two phase locking protocol. The basic two phase locking is used on the actual nodes that means the each node will run a two phase locking for the data items it is actually accessing but this two

(Refer Slide Time: 11:44)



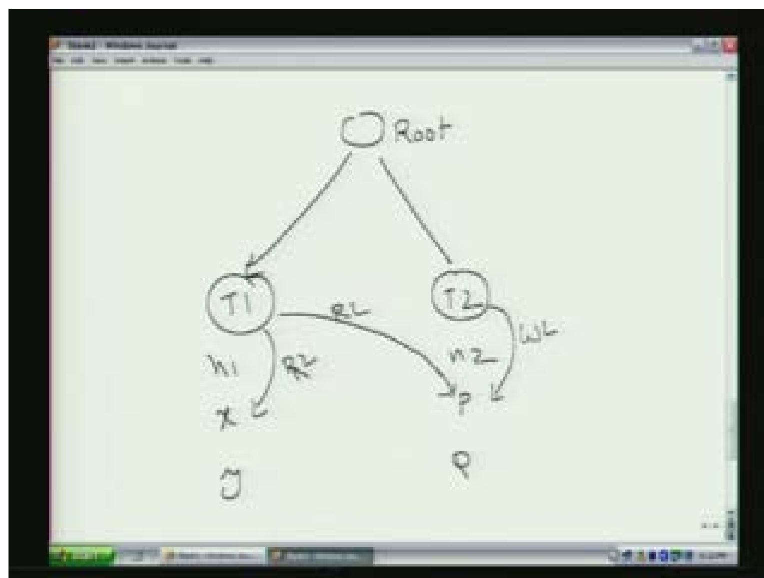
phase locking is used for local data items. You are not further looking at Tr 1 accessing a data item that is remote to that particular node and if you basically want to look at that model, you will realize that there could be problems of applying two phase locking where each sub transaction could access both local data items as well as remote data items.

(Refer Slide Time: 12:06)



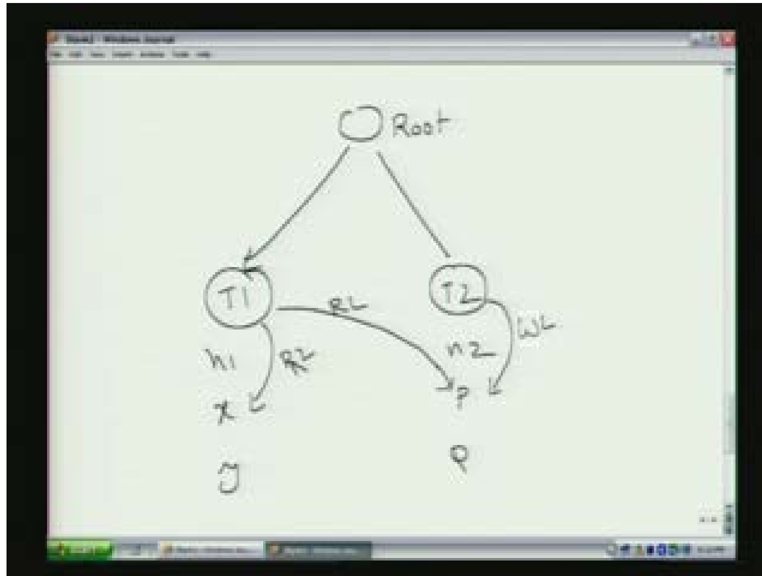
Assume for example, you have a root transaction here. This is the root transaction and then you basically have two sub transaction T 1 and T 2 here correspond on two different nodes, n1 and n2. Now let us say n one as actually data items x y and let us say this is having p and q. Now assume that T 1 actually wants to acquire a lock on x a read lock on x and then also want to acquire a read lock on let us say p, remote lock on the now which means it has the send request lock to the other node n 2. Now for example this is already locked by T 2. You have actually known T 2 has actually acquired a write lock already on p.

(Refer Slide Time: 13:22)



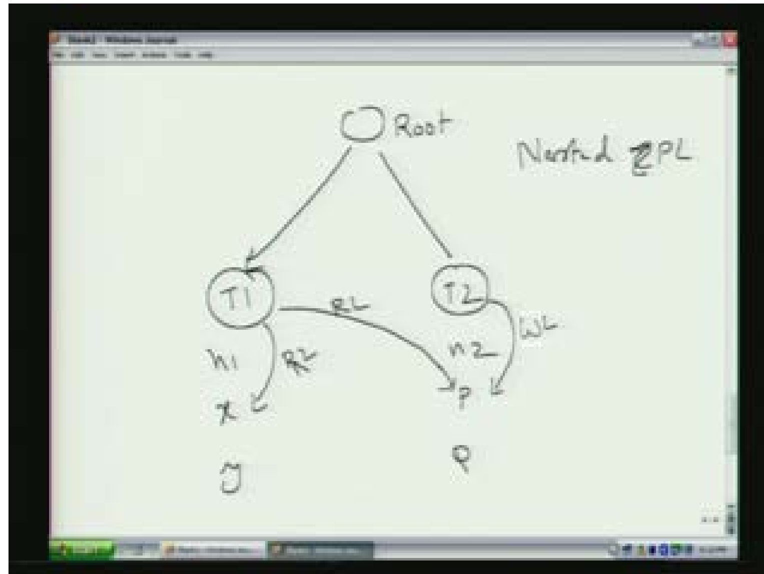
Now unless you have understand T 1 and T 2 are the part of the same transaction T 2 will be blocking T 1 from acquiring a lock on p. Normally, what you do in the distributed transaction model is to access the local data items only you will actually spawn a sub transaction here. You would not actually allow a sub transaction running on one node.

(Refer Slide Time: 13:49)



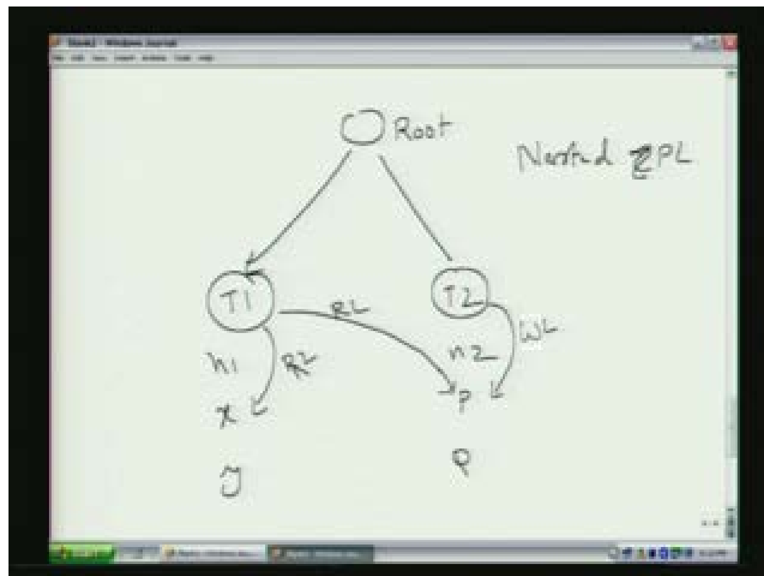
You acquire locks on basically any time you required to access a data item on a different node you spawn an agent there which is basically a sub transaction of your main transaction. Basically, if you want to provide this extension, there is an extension that is possible for a simple two phase locking which is the nested two PL.

(Refer Slide Time: 14:29)



The extension two PL takes care of the extension of the two PL to the scenario of distributed transactions where all this transaction sub transactions are seen as children of the root transaction. Now any lock requires that basically is needed by the transaction is the lock request is sent to the root node and it is basically the root node which acquires the locks on behalf of the children node.

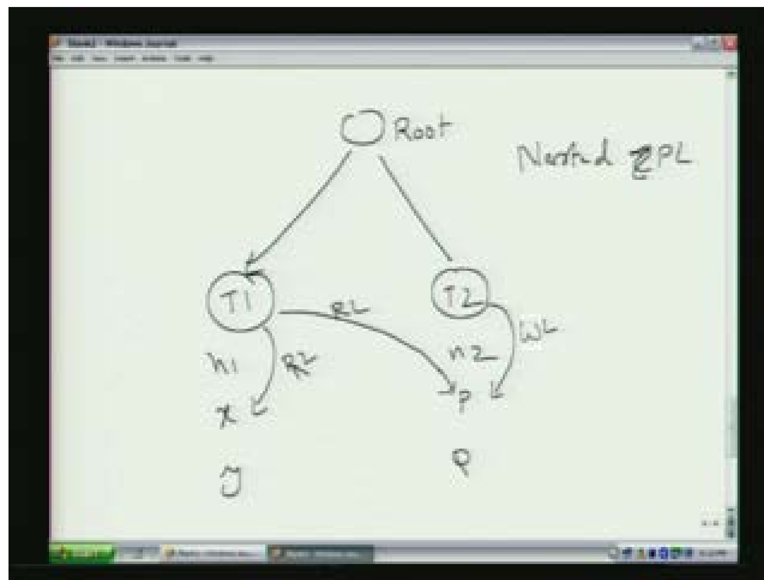
(Refer Slide Time: 15:01)



This could mean that for example: the T 2 actually wants a lock on P 2 P actually. This lock is acquired in terms of the root node and then inherited by the T 2. Now when T 2

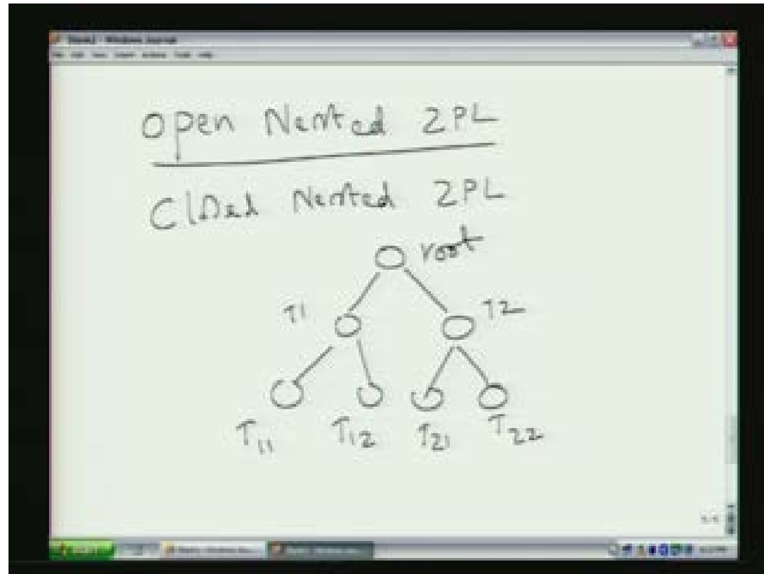
actually finishes, this lock is not released but then given back to the root node. Now if T 1 actually wants the lock, it gets it from the root node. So, all data items the locks for the data items are with the root node. Now any time you actually require a lock you are making a request to your root node and acquiring a lock and when you actually finish you actually locking the root. The root acquires this lock back. Now if these are actually the now till the same transaction the locks are held by the root. So, it basically it does not conflict the two transactions child transactions of the same root node does not basically conflict with each other.

(Refer Slide Time: 16:29)



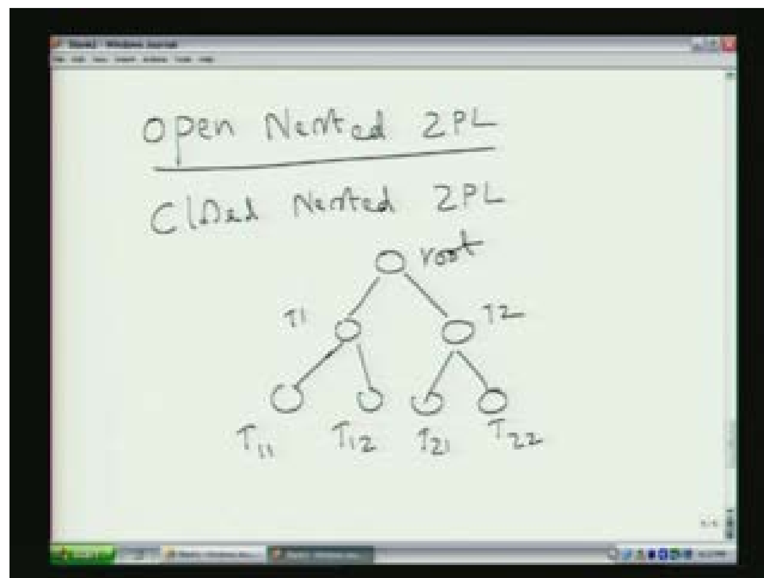
The conflict is relived because they are children of the same node. Now the way this execution is going to be done in the nested two PL is there are two kind of models that one can think of. One is called the open nested two PL and other is the closed nested two PL. Now in the case of open nested two PL, we are actually assuming that in the tree structure that you have, these are all actually sub transactions that are possible. Now, this is basically the root, can take this is as T 1 and this is T 2 and there actually two sub transactions for T 1 and they actually two more for T 21 T 22. So basically you have a tree in terms of how showing the transactions are spawn.

(Refer Slide Time: 17:29)



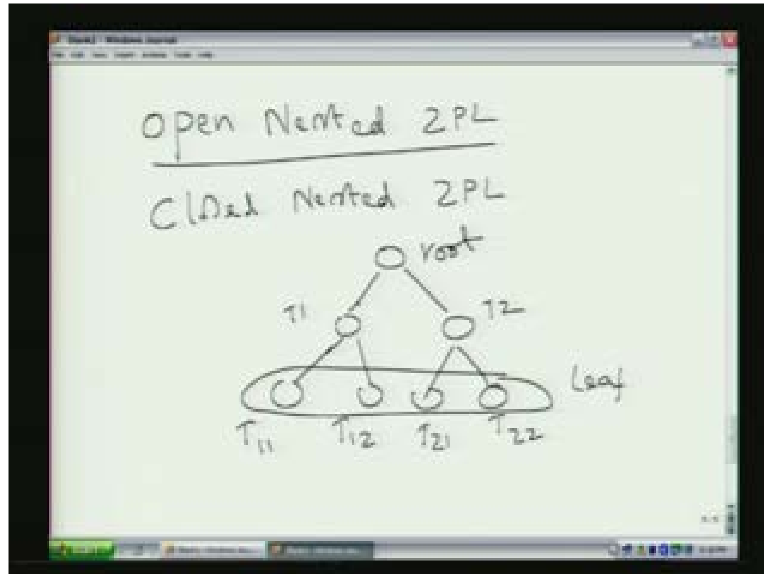
Now it is possible for only the child nodes to acquire the locks in terms of actually operating on the data item, that means T 1 will never do anything except saying that there are T11 T12 which basically access or modify the data item that means only the leaf nodes,

(Refer slide time: 18:00)



One possibility is to say only the leaf nodes are allowed to modify or process data item and if you allow these intermediate nodes also to do this processing, then it becomes open nested 2 PL and if you put the restriction saying that only the leaf nodes can do the processing on the data items, it becomes a closed nested 2 PL.

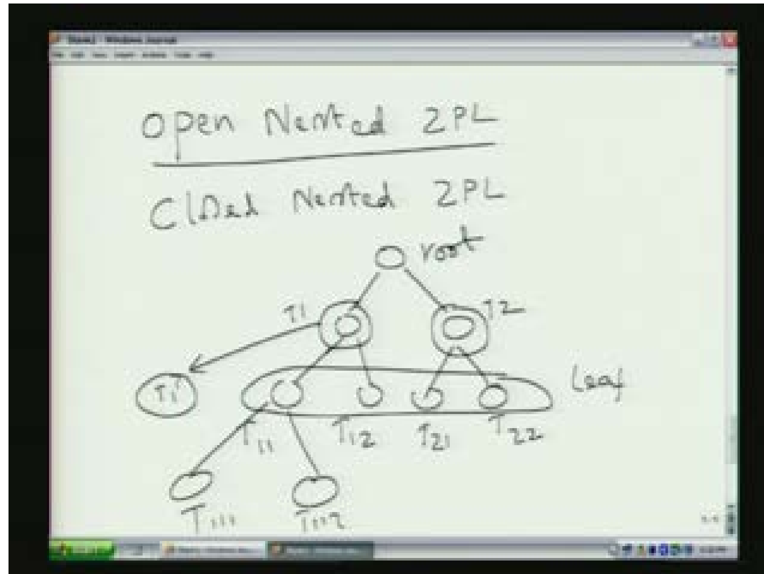
(Refer Slide Time: 18:20)



Typically, what the nested transactions are meaning here is T 1 consist of T 11 and T 22 and it is possible for T11 to further go down and have T 11 and then this can be you can further have T 112. This is the sub transaction of T 11. You further have nesting which means that you can actually make this nesting, further nesting which means that you can always divide the main transaction in to a sub transaction.

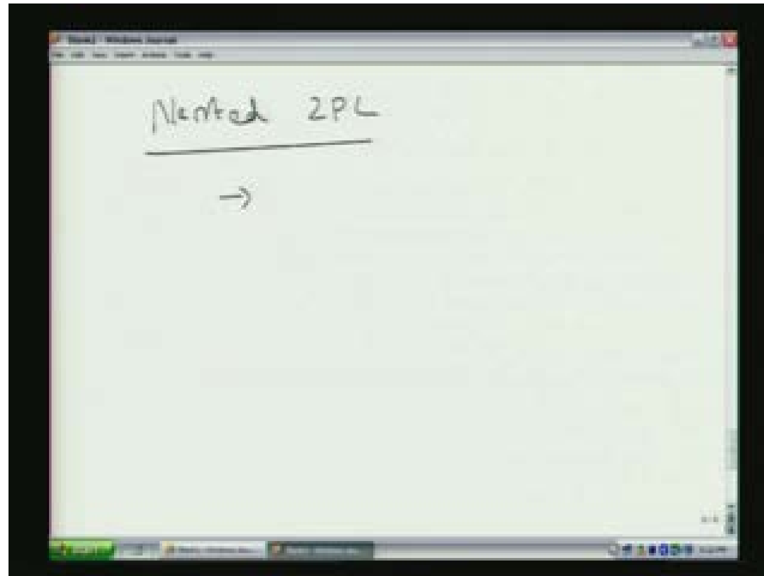
Now in terms of an open nested transaction, it is possible to actually covert an open nested transaction model in to a closed nested transaction model, because if there is processing being done at T 1, you can typically make that. For example: This can be now here spawn separately as a sub transaction of T 1 dash which means that the processing is never done at the intermediate node.

(Refer Slide Time: 20:15)



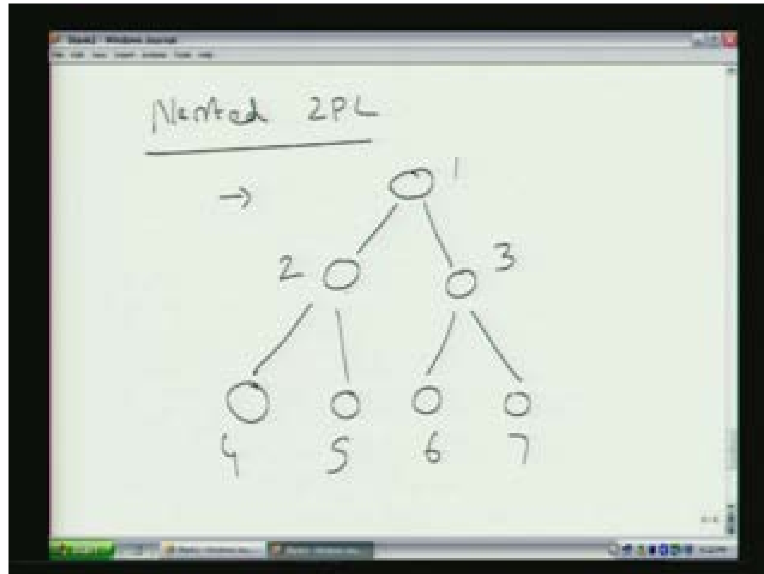
If there is a processing involved, you spawn a sub transaction to do it that means in terms of the root other than the leaf nodes, everybody else actually is spawning sub transactions and whatever the sub transaction do that result is being aggregated at the higher nodes. They do not do any processing by themselves. A simple extension of nested two PL two PL to nested two PL requires that, whatever locks are acquired by the children for doing processing are passed on to the parent when they finish execution of the transaction and it is easy to see why nested two PL will obey the serializability condition. All that that has nested two PL has is that the children never release the locks but they are inherited by the parent when the transaction finishes. Now, it is possible for another child to acquire these locks from its parents that means when you need a log, you have to first see whether your parent has the lock, then you get the lock from the parent.

(Refer Slide Time: 21:28)



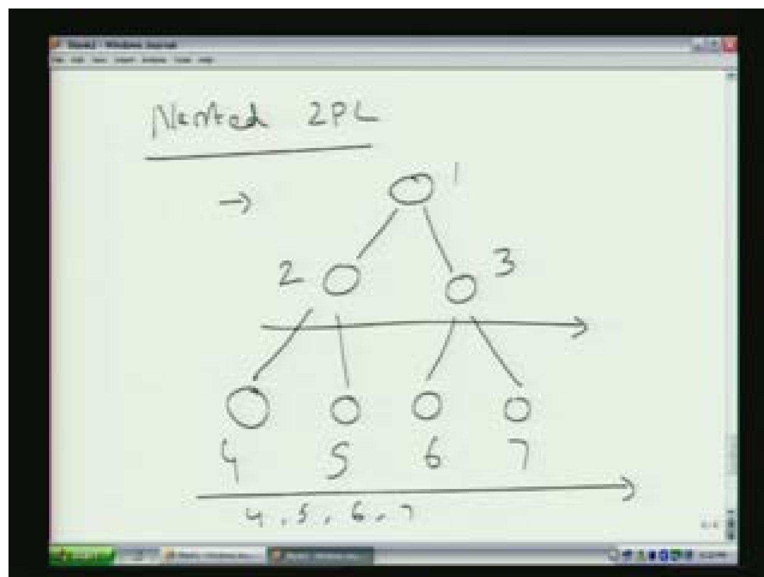
You are inheriting from the parent the lock. We assume that all the locks are with the parent. So the only way you can acquire the lock is by making a request to your parent. When you actually finish your execution, you are returning the lock back to your parent. Now you can see why this simple nested two PL will ensure serializability by looking at intuitively what is happening here. Now, if you basically see that there are two branches of the tree and there is a lock acquired by any of this children here, there is now an inherent no serializability at the higher level because unless let me put in terms of simple tree structure nodes. One, two, three, four, five, six, seven. Now you can see that the inherent order in which this execution can proceed is, if three requires a lock or six requires a lock that is currently held by any of the other nodes the only way it can be obtained is parents which actually means that unless two finishes.

(Refer Slide Time: 22:51)



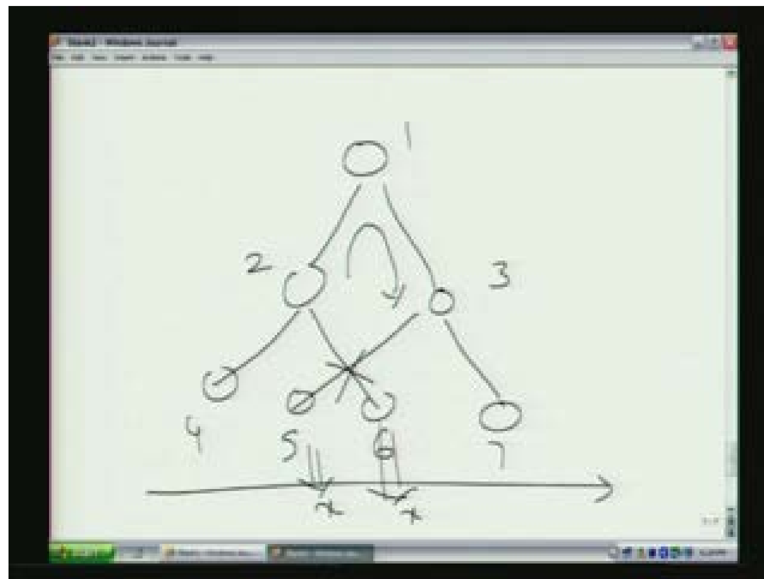
One cannot acquire the lock that means at the higher level the execution is always proceeds in terms of this tree, the nodes of finishing one node is which is earlier till finishes and later the other node can finish. Otherwise, there is no way other node than finish. So inherently that tree structure will be ordering the way the transactions are executed and inherently provides a serializability condition. Now you can say that, that means the only execution sequence that is possible in the case is four after that five if there is a conflict, then six and seven.

(Refer Slide Time: 23:38)



If there is no conflict, it is possible for this tree structure to be executed. For example: where the leaf nodes are all. So that the same tree structure that we saw earlier it is possible that, i can have execution where if there is no conflict, because if there is no conflict, it does not really matter how six and seven are actually executed. If there is a conflict that means, if five is asking for lock on data item x and six is also asking for the data item x, this wont be possible because this lock would have been held by node unless it finishes execution, this would not have happen and hence this will be prevented. You can see in a simple case, a nested two PL is an extension of a simple two PL where a transaction actually spawns a sub transaction.

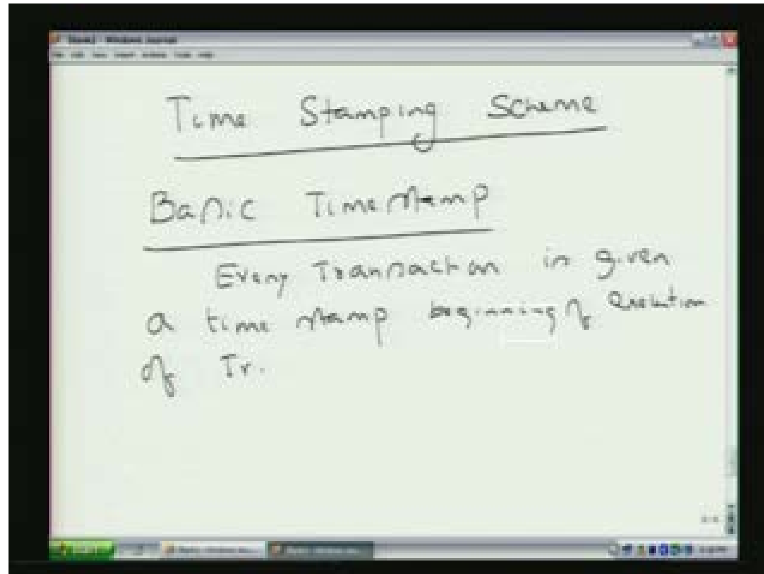
(Refer slide time 24.43)



and now that sub transaction can further spawn other sub transactions and then each sub transaction when it finishes gives a lock to the parent and that is the way to execution can proceed and that inherently ensure is the serializability requirement. This is an extension of simple 2 PL to nested 2 PL for distributed systems. This is one possible model. You also seen a number of actually time stamping scheme as part of our earlier discussion on transaction models. Now it is worth actually seeing what exactly happens with time stamping schemes in their distributed scenario.

Now what we have is, the basic time stamping scheme which shows how exactly transactions can be executed by having a basic time stamp being given to both the transactions as well as to the data items. Now in the basic stamping scheme every transaction is given a time stamp at the beginning of the execution. Now this time stamp could be just the logical lock value that we saw in the distributed systems.

(Refer Slide Time: 26:47)

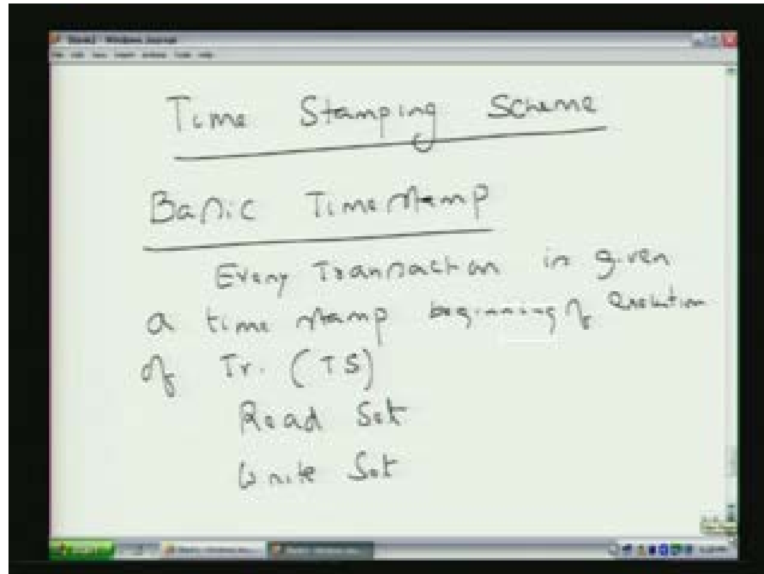


which means that this is a local clock that is maintained by the distributed system which in affect means that this is to be either synchronized clock or some kind of logical synchronization happens these clocks such that the time stamp reflect the total order among the transaction that have actually executed in the distributed system. One way to do this is the last two bits, know least significant bits a part of each node and the higher bit is actually a synchronized bit with the other clocks of the other nodes.

It is possible for you to give a time stamp for each of the time transaction originating in each of the nodes of the distributed system. Now, if you assume that every transaction entering the distributed system can be executed on any node of a distributed system, so this time stamp corresponds to the logical time that we saw in terms of the different node synchronizing their logical clocks. One extension is a simple case of mutual exclusion algorithm that we had implemented in the case of lamport's clock. The same scheme is applicable here except that instead of mutual exclusion is going to have a read lock or write lock that you acquired on the data item that means in affect this is equal to the time stamp being used for writing on the data item or reading on the data item.

Now different nodes will decide the order of the in which they can access this data item by using the time stamp value. The basic time stamping scheme what you are essentially doing is you are actually giving time stamp at the start of the execution of the transaction. Now as the transaction is start executing your actually looking at its read set and you are seeing its write set and then you are deciding in terms of the read set and the write set, what are the time stamps of these data items and making sure that the transactions are executing in terms of the time stamp order. Now, assume that every read and write data items in the data base have given the time stamp value.

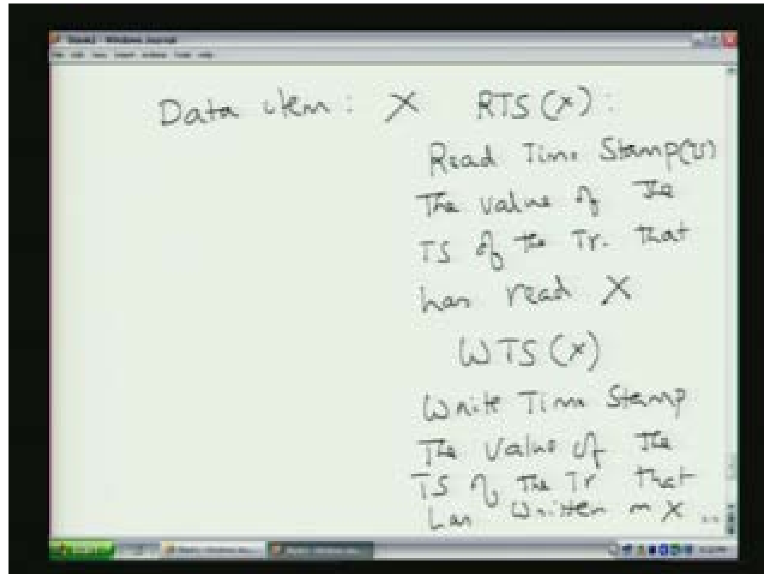
(Refer Slide Time: 29:54)



For example: Let us take a simple case of actually looking at a data item x which is in the database. Now each data item will have two time stamps corresponding to it. One is the read time stamp that means this gives basically the read time stamp. What the read time stamp will you give is the value of the timestamp, the highest value of the timestamp of the transaction that has read x that means it tells, which is the highest time stamp transaction that has read its value. Now similarly we will have one of write time stamp X which tells correspondingly the write time stamp of the transaction.

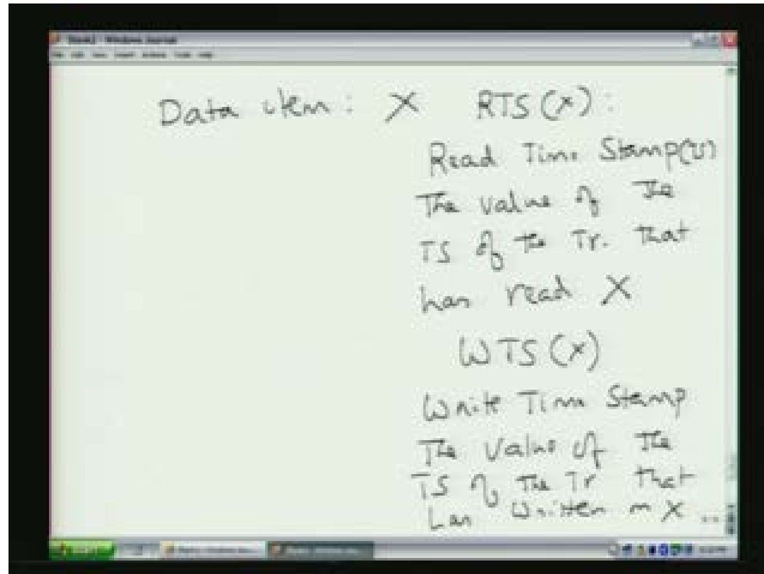
This is basically the write time stamp. The explanation for the write stamp is similar except that instead of what we have put there as read. Now this is the highest value of transaction which has actually written on the data item X. The value of the time stamp of the transactions, that has return on X.

(Refer Slide Time: 32:04)



Now the use of the read of time stamp and the write time stamp is for serializability, you always allow the transactions to only do in terms of increasing values that means if there is new transaction that comes in whose time stamp is lower than the write time stamp and if you want allow to read the value you obviously will have problem right. For example: If the write time stamp of the data item is greater than the transaction time stamp, then you should prevent it from reading the x value that is basically you to abort the transaction and then ask it to come again with the higher time stamp. So that you are preventing the access to the data item in terms of always the data items should be access by the transaction in terms of increasing time stamp. If you ensure that you are automatically achieving the serializability condition.

(Refer Slide Time: 33:20)



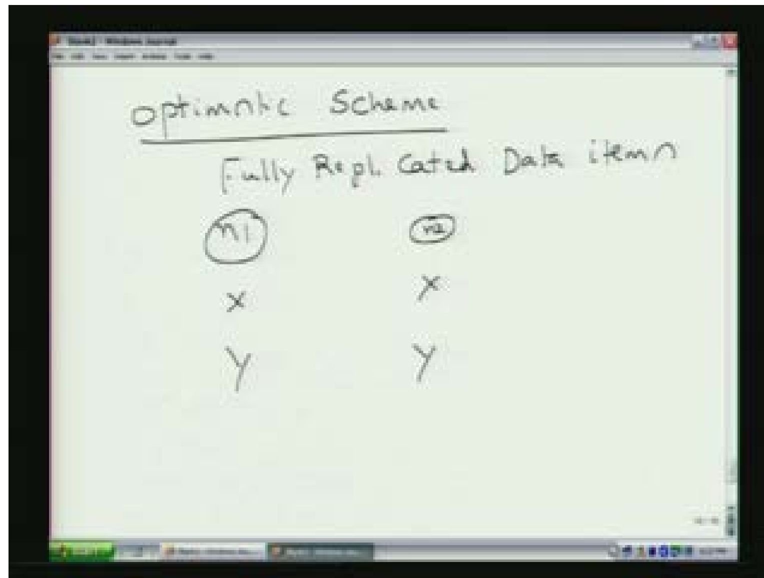
Now in a simple way in a distributed setting you are going to extend the basic time stamp distributed transaction model by ensuring that these time stamps are generated in a consistent way by the nodes of the distributed system. If each one of them generate the time stamps the transaction time stamp are generated in a global way even when the data items are distributed you are still ensuring that globally all the reads and writes are ordered according to the time stamps of the transaction. Since the time stamps of the transactions are generated in a consistent way in the global system, we are essentially ensuring a serializability order for your transaction coming in the distributed execution. Basically that is an extension of basic time stamp scheme to the distributed scenario the extension is quite simple and straight forward again in this particular case ensuring that the logical clocks of the distributed system are synchronized and executed properly as per that order.

Now one of the interesting extensions that one can see is, a completely optimistic extension of the time stamping scheme were the data is fully replicated on all the nodes which means that now every node has a data item available locally. So what you are going to do now is, you are going to look at each node is going to send its read and the write set to everybody and now everybody can vote on the reads and the writes and if whatever reads and writes you have appropriate number of votes then you will be able to view the modification which means that the majority of copies are always consistent and always proceed in the same order as being done by the nodes.

Let us look at that extension and is often called the optimistic time stamping scheme and i will actually summarize the different time stamping schemes which have done earlier with respect to the distributed scenario. The optimistic scheme is interesting extension because what you have is here a fully replicated database that means the data items are fully replicated on all nodes. Now take a simple case were there is node n 1 and there are two data items x and y and take another node n 2. What is going to happen now is, it will

also have x and y which is essentially means that all that a transaction is going to do now is going to produce you known a list of data items that is willing to modify along with the transaction time stamp which is trying to do.

(Refer Slide Time: 37:00)



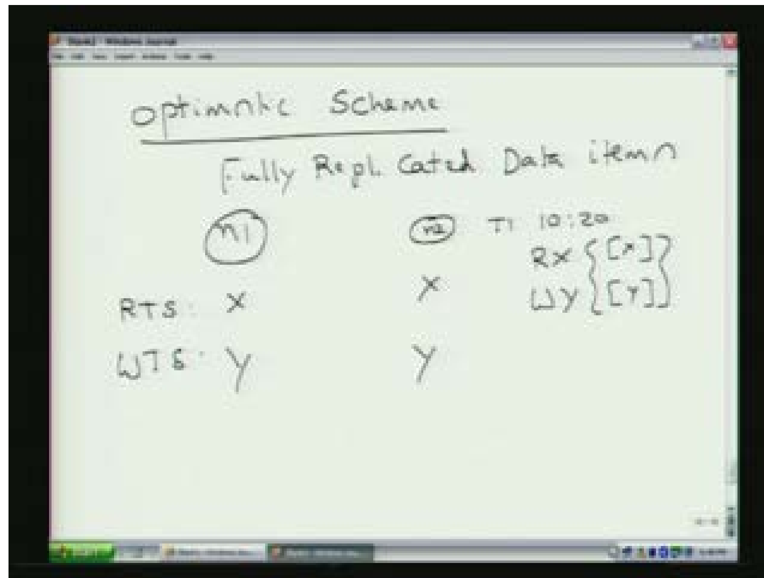
Now what is going to do now is, it needs to send the information to all the other nodes and other nodes have to now say whether they are willing to accept the list sent by another node, that means every transactions modification what is read what is to modify will be send to all the other nodes and now those nodes can decide whether that update list is consistent as far as they are concerned. If they say yes, then if enough number of yes votes are come for a transaction then obviously can commit. It does not get that it does not basically commit which means all the processing always going to happen locally but then you have to send your read list and write list for voting to other nodes.

Now the other nodes can decide to vote yes in this particular data item. Let us take a little more detailed way and then see what really happens in this particular case and how the transaction execution is going to proceed. In this particular case, what we going to see is let us say i have a transaction t 1 which started at let us say ten twenty here with respect to its time stamp. Now its basically trying to read x and write y. So that means its read set consist of x data item and it is basically write set consist of y. Now it will modify whatever it wants to modify as far as this transaction is concerned and produces this list for other transaction to vote, that means it will communicate this list to node n 1 saying that as far as this t one is concerned, it is at time ten twenty it is modifying x and y and that list is being sent to the other node.

Now assume that, there is basically a read time stamp on x which actually shows again, what is the highest value of the transaction that has actually read x and similarly the write transaction on y. Now what you are going to see is, if this list is actually non conflicting as far as the other node is concerned, what we mean by non conflicting is basically there

is set of transactions want to execute on other node as well. Now what you have to check on the other node is before the modification is done whether this modification is consistent as far as the other node is concerned and the consistency criterion is here is, this x and y whatever this transaction is trying to do now is later than whatever has been done already in the other node.

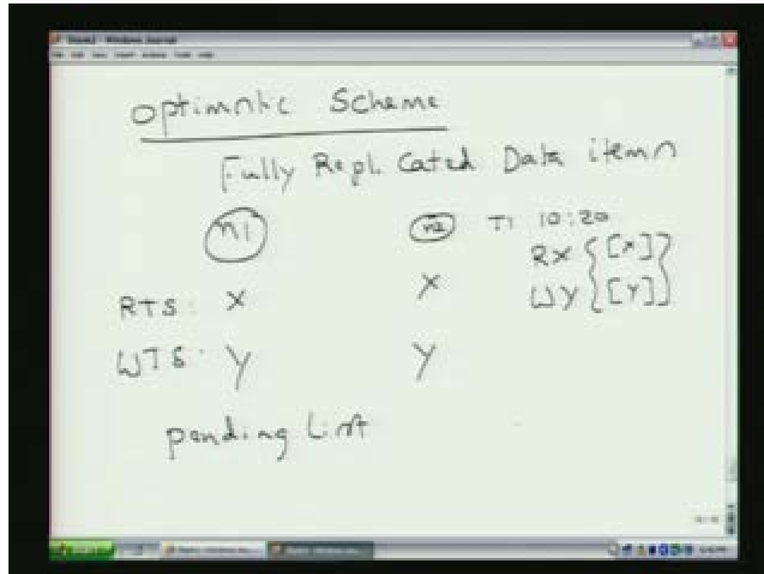
(Refer Slide Time: 40:22)



which means that the corresponding RTS and WTS tools are applied on the data item and if it is consistent as far as that is concerned then you have vote saying that, this is okay for me because as far as i am concerned, there is no conflict of this transaction for serializability and if it is basically conflicting and this condition does not hold good the RST or the WTS not being know in the case of write transaction, it has to be more than both RTS and WTS. In the case of read transaction, it should be more than the write transaction WTS. So, in that sense we apply the consistency rules of RTS and WTS on the other node and decide whether this list is consistent.

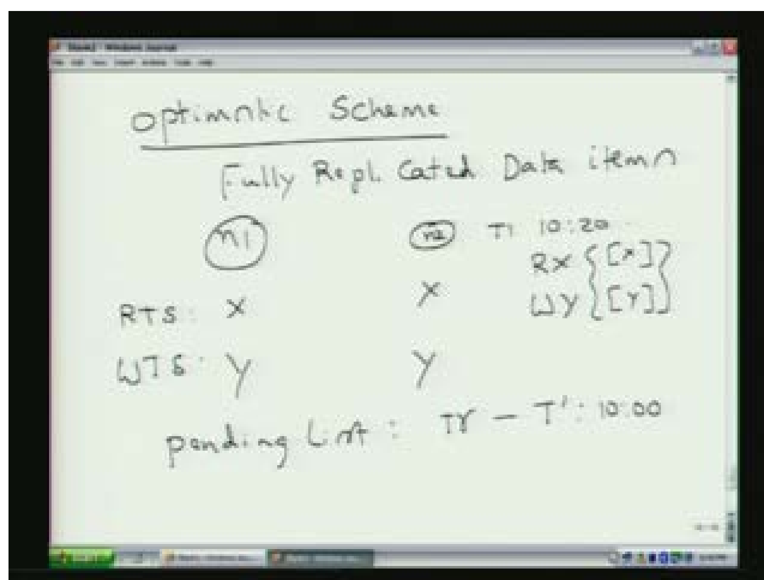
Now that would not be enough because this is only a proposal to modify. It is still not modify. They could be several such list come to me. Now it is possible that there is a pending list with me and i have to actually compare these values with the pending list also, what is pending now with me? Now let us say in the pending list, none of it actually conflicts with what i have then the rule is simple. Then I will be basically give a ‘yes’ vote for it.

(Refer Slide Time: 42:02)



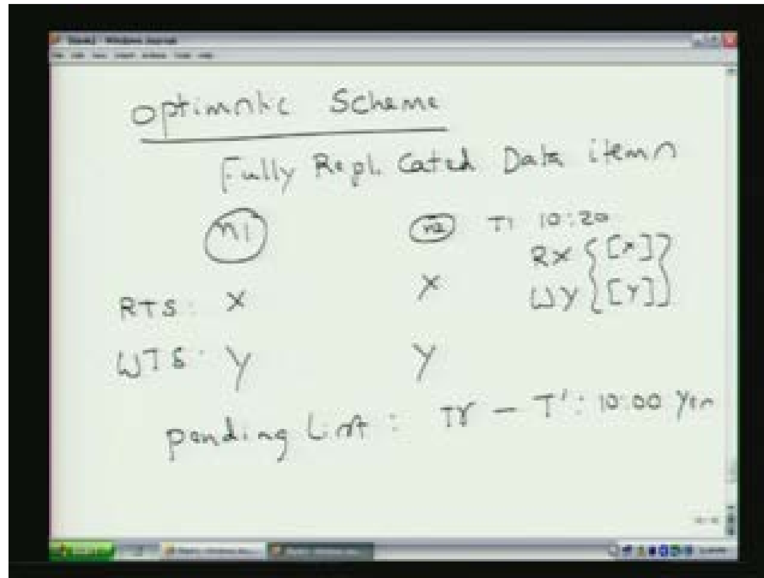
If there is a conflict in the pending list, which means that there is a list which came to me which i has actually voted. Now let us say there is a conflicting list with me and in the conflicting list, i have voted and yes for the conflicting list. Now, there are two things that can happen with the conflicting list. Let us say there is another transaction whose time stamp value is let us say ten twenty now. There is a conflicting T dash direction here whose value is let us say ten o clock time stamp value is ten o clock and let us say for this obviously I have voted and as earlier that is why it is actually my pending list. I have still not heard about this transaction.

(Refer Slide Time: 43:08)



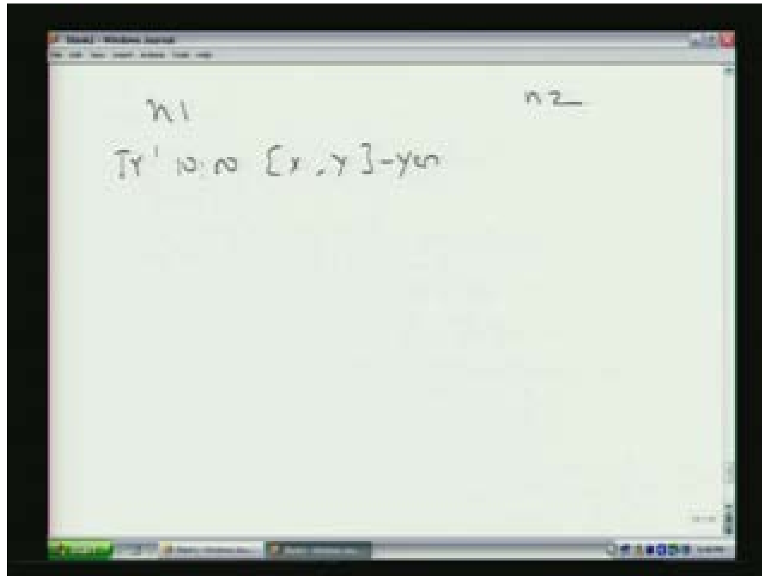
Now one of the things, i can do in this particular case is since i have already voted for t 1 as 'yes' T dash as 'yes'. The best thing i can do for t 1 is say 'no' because it is conflicting and an earlier transaction already said yes. Since i have actually said yes to an earlier conflicting transaction which means that this t dash has actually 'yes' for my case. Now i actually another transaction at t 1 which is actually conflicting with my pending list. Let us say this x value trying to modify and when it is trying to modify the value of x probably if we take a specific example here to make this little more clearer.

(Refer Slide Time: 44:03)



Basically what you are saying is this node n 1 when it got a list, it is voting 'yes' or 'no' right. Now let us say there is basically let us take node n 1 the execution sequence, the node n 2 the execution sequence. It received an update list of Tr dash at ten o clock and this is the list with x and y let us say, it receives this is the update list. Now, it did not conflict with anything by actually voted for yes. It still pending on my side because i have not heard the final decision on this, because the one who has actually sent me the list if it get enough yes votes then it is going to get committed and send it finally to me as

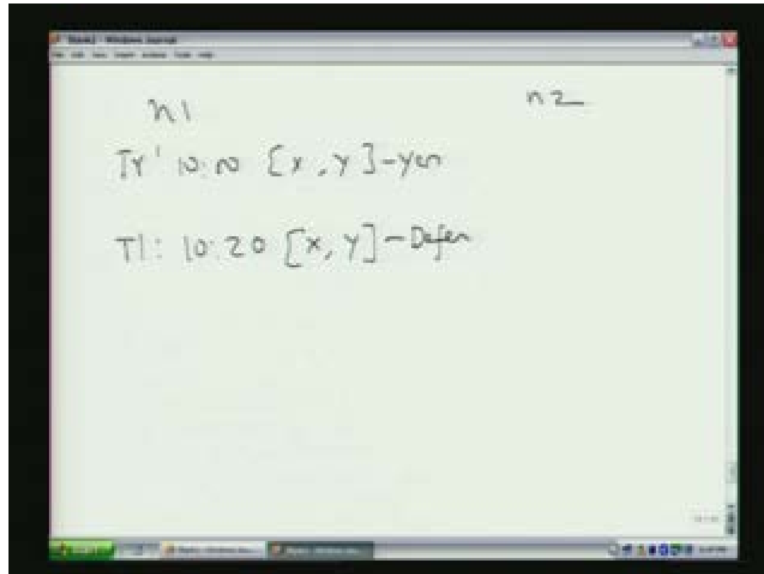
(Refer Slide Time: 45:01)



an update list, then will apply the updates to the actual that is my update. But, in this particular case this is the pending list. I have voted but then the decision on this still pending. There is no decision on this particular list. Now what happened? There is another transaction t 1 that came by and it is actually ten twenty and then now if it is not conflicting with my pending list, there is no problem what so ever then i basically saying 'yes'. Now let us say the one which came which came at ten twenty as a pending with my conflict pending list. Now for that pending list, I have actually voted as yes.

Now one way for me to do is, this ten twenty i would not reply immediately but wait i say okay till i get the decision on ten o clock, i am not going to vote ten twenty which means that differ making a decision on that is one possibility. That means this list is not voted by me. I am not going to say yes for it a 'no' does not create a problem because a 'no' is already been. I have actually said i am not willing to go with that pending list.

(Refer Slide Time: 46:31)



So even if it conflicts no really does not bother you because you have actually voted but some other reason you actually not willing to go with that transaction. Only when you say yes to something, then the current one is conflicting with it, then you have you have a problem now whether will this create the problem or will not create this problem because based on that you have to take decision. Now assume a scenario where this is this scenario 1. This scenario 2 could be same T 1 the same T 1 arrived at my node and now it is conflicting. Now here also let us say, i take the decision in to differ making a decision in this case.

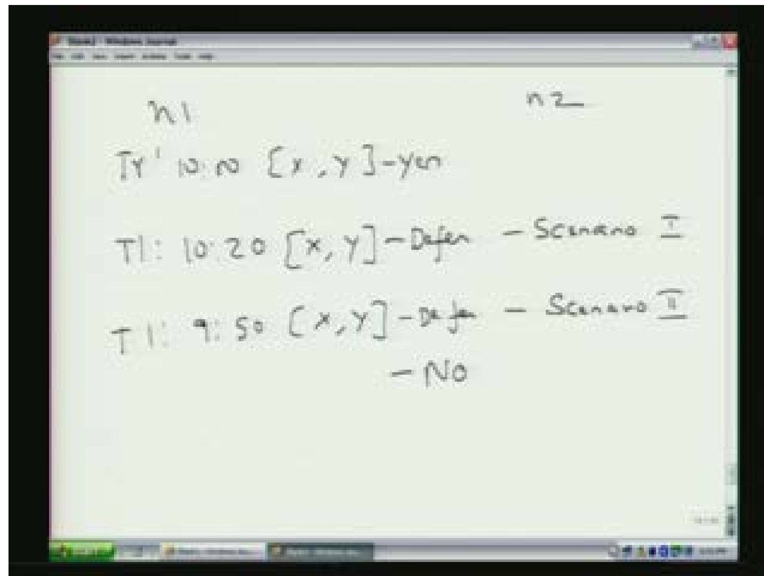
Now both scenario 1 and scenario 2 differ, then imagine a possibility were none of the list the likely would have getting a majority may not be there because everybody is waiting for the list to be voted and there could be the case were none of them might receive enough votes and everybody is waiting for everybody else. Suddenly, it looks like differing voting on both cases is not a good idea. Now the next case which case i better vote no and which case i should better vote ‘yes’. Now, if you basically looked at i have already voted at for that ten o clock case is yes.

The best bet for me is that ten twenty can be differing because it is the later transaction. Suddenly if i have done something at ten o clock and somebody is trying to come to do at ten twenty it is still alright with me, but if it is actually nine fifty, i better in this particular case vote as no because i have already this any way will be only one chance it would have succeeded is the case were ten o clock did not get enough votes. In that particular case, the T 1 has a chance.

Otherwise it does not have a chance right. So i can wait. I can always wait both cases, then decide only when i know about the know fate of ten o clock transaction because i have voted for it and make any future things that have coming in. Till the pending list is clear i would make any other decision but what happened in the case is the possibility of

all the transactions waiting for each other pending list and result in no body progressive. So you better vote the case where it is nine fifty as you vote for this as a no to avoid this confusion.

(Refer Slide Time: 50:14)



Now at the end of it, the majority votes if a pending list actually majority vote that basically sent as an update list for all the nodes, that means that list is now has to be updated by all the nodes. Now the very fact any node has got enough votes any pending list has got enough votes that becomes a permanent write on it and you can easily show that no to conflicting things can get.

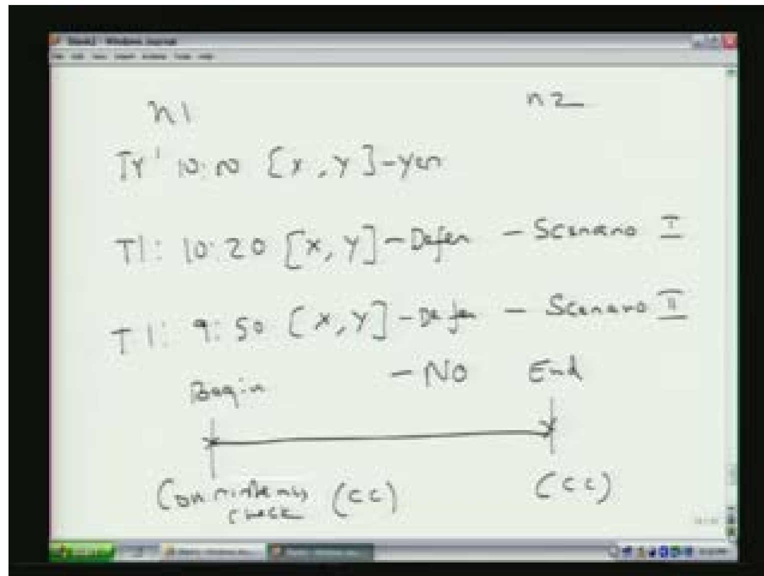
Both cannot get majority vote at the same time because only one of them can win if the two transactions are conflicting, only one of them in the end will be able to achieve the majority vote and that is the one which will go through and the other is going to be restarted. If you basically look at it here, it is a completely optimistic scenario because you are executing all the transactions and you are never checking the consistency before start execution of the transaction.

What you are doing is you are executing the transaction and at the end of it, you are deciding by sending this updates to all other nodes. You are seeing whether you can go through with this execution or not. So this is typically the pessimistic scenario verses the optimistic scenario. Pessimistic as we told earlier, the consistency check is made at the beginning of the execution and then this is the begin that means actually you do this checking a locking would have ensure that the check is done in the beginning unless you acquire the locks will not proceeding on execution.

On the other hand, this check is made at the end which actually means that we are actually executed in transaction fully and then applying the consistency check. Now in that sense, basically having the two spectrums. One is the locking based algorithms

which fall in this spectrum and then a fully optimistic time stamping scheme which for in the scheme were the consistency check is only applied at the end of the execution sequence. A variety of you know models are possible for as the transactions a models are consistent based on the time stamp scheme.

(Refer Slide Time: 53:10)

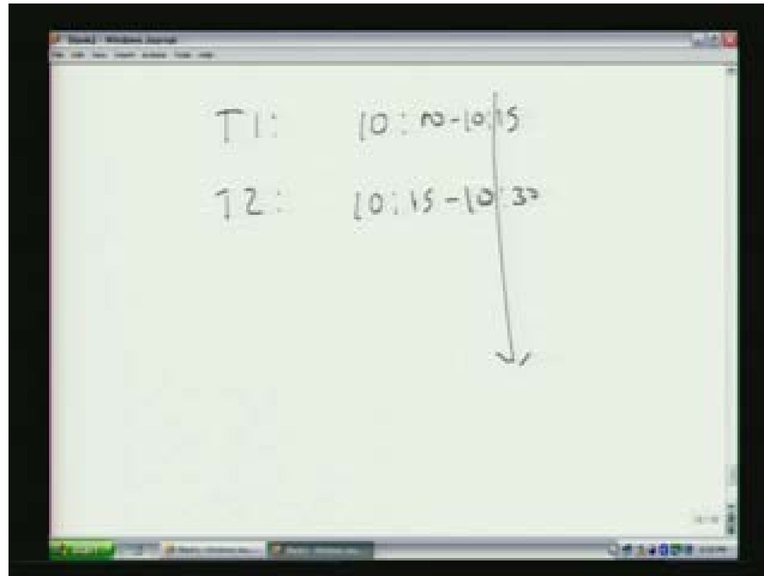


One of the most interesting scheme is to give what we have seen in the case of example: Visiting Tirupathi temple. You have the model were the band of time is given for you; you have to go and then see you know thing at the point of time. Basically, you are scheduling transaction which can be executed in the future. For example: you can say as far as t 1 is concerned, you can give a band of time which it can really execute which means that when it start executing you can say that, this becomes the time before you which commits it is alright for you.

This is like telling, if it comes at the gate at ten twenty. This is not the start time, virtual time of the commit as far as the transaction is concerned, if it commits at let us say ten o clock its fine with me yours telling when it commit and it can actually finishes it execution. So you can actually give an order in which, this is basically end finish time. This is not at the start time that you are trying to schedule. You can actually do this at the end. For example: it is possible for you to tell even a band.

Let us say, this transaction I can allow between 10 and 10.15 to execute, finish its execution. If i generate this band such that, conflicting transactions will not finish in the same band I have actually know it is getting the serializability condition. For example, there could be another. For example: if the band that is given to these two transactions is different, if they are conflicting.

(Refer Slide Time: 55:01)

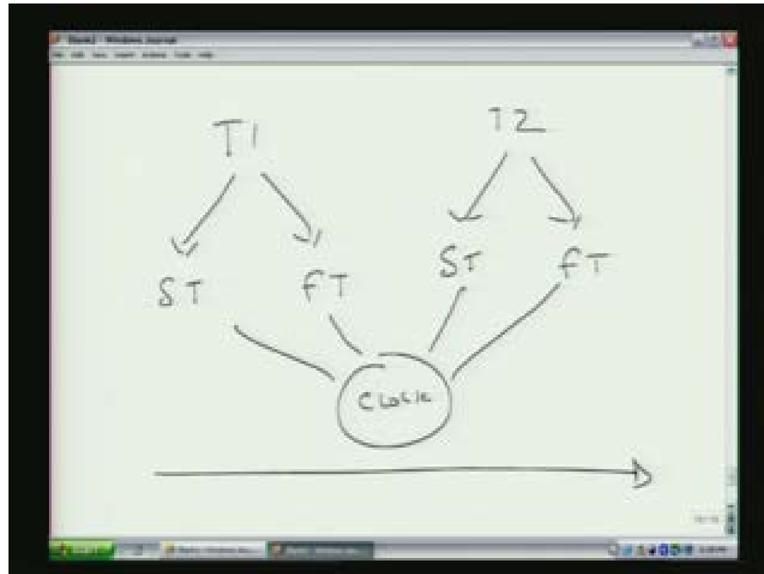


Let us say, i give a band of 10.15 to 10.30 for this. It actually means that, suddenly they are not you know the serializabililty condition will not be violated here because one is finishing at 10.15 and other is finishing. So i actually gave the band of time in which they can finish their execution that is other model i can give. Completely different kind of model is also possible were you can say that I will basically do what is called only transaction time stamps, but not data time stamps. For example: all the schemes which i have seen so far have two kinds of time stamps. One is time stamp to the transaction and i am giving the time stamp to the data item. You could do a scheme were all that you are going to do is as the transaction is executing you can give time stamps to different points of execution.

For example: you can give start of the transaction finish of the transaction, what is the time and used that for actually doing the consistency check but as long as one transaction finish after other their conflicting and that you are able to ensuring at your clocks times that you have actually happened. You are actually able to ensure the consistency condition. In a simple way what we are talking about is, let us say T 1. There is a start time and there is a finish time of T 1 and similarly T 2.

There is a start time and finish time and these times are actually given by using a clock here. Now, as long as the conflicting transactions are all as they are executing these times are given, you can ensure that the serializability condition as far as the operation is concerned. By making sure that, the start and the finish time of the properly synchronized as far as the transaction execution concerned. By doing this, we actually do not need to maintain time stamp on the data item.

(Refer Slide Time: 57:27)



That is a saving in terms of not needing to maintain the time stamp. A variety of schemes are possible we have seen couple of scheme in this particular lecture.