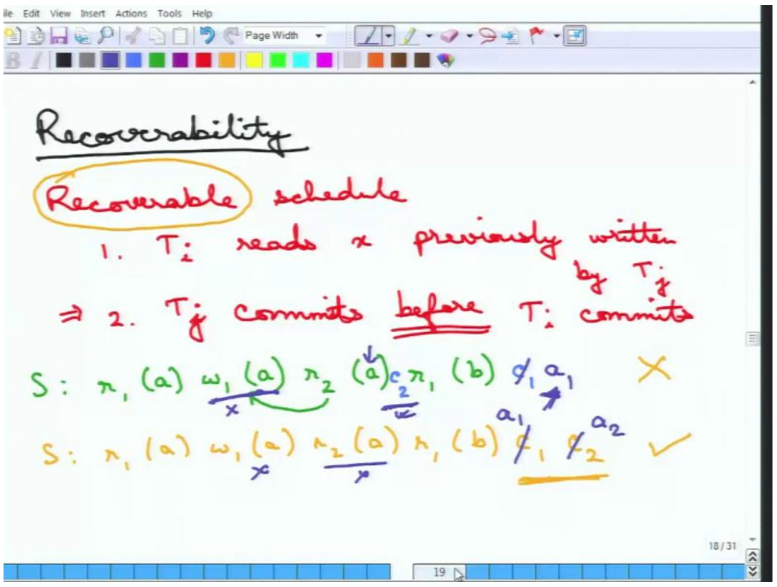
Fundamentals of Database Systems
Prof. Arnab Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Kanpur

Lecture - 37
Schedules: Recoverability

So, we will next move on to another important property of schedules, which is the recoverability. Now, we have been studying in the serializability or in the conflict serializability and the view serializability is that, whether they are equivalent to each other. Assuming that all the transactions in the schedule are correct; that means, all the operations are successfully done, processed.

Now, what happens in most transaction, it may happen that the order, the transaction may abort etcetera. So, whether the transactions can recover or not; that is the study of recoverability. So, essentially in the serializability study, we did not take into account the commits and aborts and the order, more importantly the order of when a transaction commits and when a transaction abort etcetera. That is what we are going to next study, so this is on recoverability of schedules.

(Refer Slide Time: 01:00)


Recoverability

Recoverable schedule

1. T_i reads x previously written by T_j

$\Rightarrow$ 2. T_j commits before T_i commits

$S: r_1(a)\ \underset{x}{\underline{w_1(a)}}\ r_2(\overset{\downarrow}{a})\ \underset{w}{\underline{c_2}}\ r_1(b)\ \cancel{c_1}\ a_1$ ✗

$S: r_1(a)\ \underset{x}{w_1(a)}\ \underset{x}{\underline{r_2(a)}}\ r_1(b)\ \overset{a_1}{\cancel{c_1}}\ \overset{a_2}{\cancel{c_2}}$ ✓

So, recoverability of schedules, so here the essential idea is that the order of commits and aborts in the transactions are important. Now, the first thing is, we will define which is something called a recoverable schedule. So, schedule is set to be recoverable, if certain conditions happen. So, recoverable schedule, essentially the definition is the following. Suppose, in the schedule the transaction T i reads our data item x; that has been previously written by some other transaction T j as part of the same schedule of course.

Now, it must happen that if this holds, then if one holds, then the second condition must be holding is that, T j must commit before T i commits. So, if this happens and in the second condition either T j commits before T i commits. Now, why is this, intrusively we can see, what is the reason for this is that, T i has rates something that has been produced by T j.

Now, unless T j really commits meaning it vouches that what it has written is correct, T i is the reading of T i may not be correct. So, in other words, suppose it happens that T j now aborts. Now, T j aborts meaning, whatever written by T j needs to be rollback and needs to be undone, which means that whatever T i read, which is the value written by T j is also incorrect, because it should not have read this, because that has been undone.

So, T i it fit at already committed, then it is in a problem, because now it has committed with a wrong value rates. So, that is why, this is not allowed and that is why are schedule is called recoverable, if these, if T j commits before T i commits, if the first condition happens. Now, let us directly jump to an example and let us see what we are trying to say. So, suppose our schedule is the following is r 1 a, then there is a w 1 a, then there is a r 2 a.

So, the transaction 2 then read and then, suppose transaction 1 read some other data. Now, essentially what is happening is that, you see that transaction 2 is reading the a which is written by transaction 1. Now, in that case, the rules is that, T j must commit before T i commit. Now, suppose what happens is that, let me introduce this thing is that suppose after this, there is a commit by c 2 here.

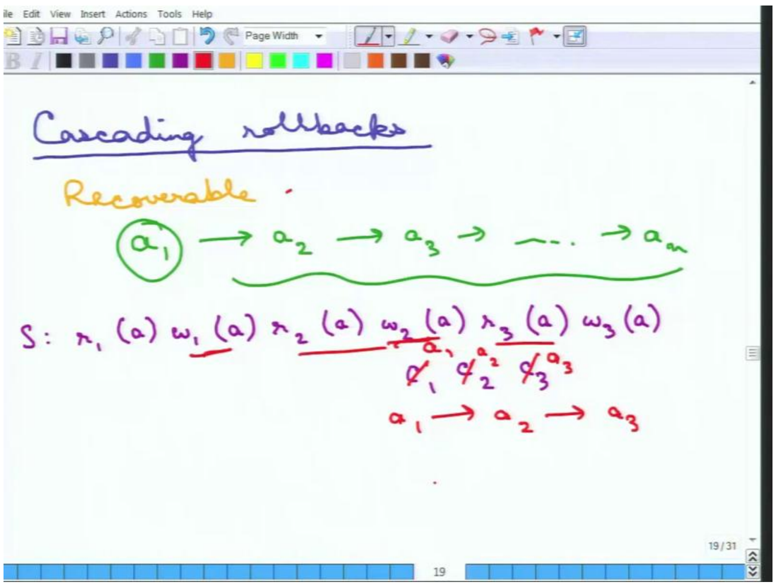
Now, the question is whether this schedule is recoverable or not, so this is not recoverable, am I assuming that c 2 commits here and then, c 1 commits here. This is not recoverable, because c 2 should not be committed before c 1 does. The reason is suppose c 1, it is not a commit here, but it is an abort by c 1. Then, what happens is that these values; that is produced is wrong; that means this rate that what it has done from w 1 is also wrong; however, it has already committed.

So, by the time the system reaches here and finds that, this is abort, this is already committed. Committed meaning, if you remember the property of durability, when a transaction commit, it essentially says that, if I am committed, then everything that I am going to do is final and correct now, but that is a problem. So, this is not recoverable, so this schedule is not recoverable.

Now, to make it recoverable, what should have been done is that if this is the case, if this is the schedule, then r 2 a, then r 1 b etcetera, what it should be done is that c 1, so c 1 transaction 1 should first commit and then transaction 2 commits. So, this order of commits is very, very important. So, that is the reason why you studied recoverability this order of commits and abort is what we are very much bothered about, because that is way a schedule may be recoverable or may not be recoverable.

So, now the reason is first is that, suppose just like a previous case, somehow this commit is not true, but this actually abort. By this abort; that means this value produced is wrong; that means, this r 2 should also be wrong, which is fine, because then the transaction 2 can then simply abort. Instead of committing, because it is read wrong value it can also just abort. So, then there is no problem as for as the recoverability of the schedule is constants. So, that is the recoverable all that seems to be fine.

(Refer Slide Time: 05:48)


Cascading rollbacks
Recoverable
$\boxed{a_1} \rightarrow a_2 \rightarrow a_3 \rightarrow \ldots \rightarrow a_n$
S: $r_1(a)\ w_1(a)\ r_2(a)\ w_2(a)\ r_3(a)\ w_3(a)$
a_1 a_2 a_3
$\cancel{c_1}$ $\cancel{c_2}$ $\cancel{c_3}$
$a_1 \rightarrow a_2 \rightarrow a_3$
19/31
19

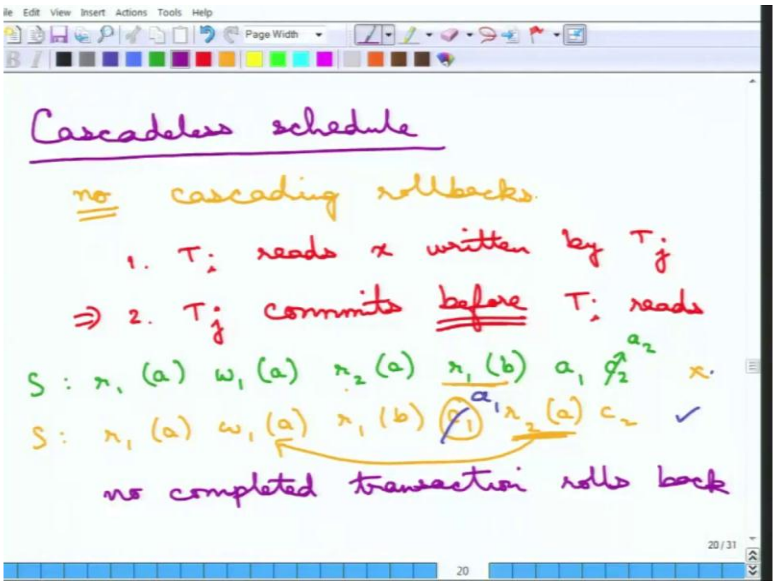
But, then it runs in to one problem, which is called a problem of cascading rollbacks, by the way, this rollback is essentially what the undoing operation etcetera is. So, when an abort happens, the transaction essentially rolls back; that means that, if undo of the values and writes the old values of etcetera, so that is the roll. Now, the problem is, even if a transaction is recoverable, there may be a series of rollback as we saw in the other the previous example, if there is an abort by transaction 1, then this leads to an another abort by transaction 2.

So, in a general sense, this may lead to an another abort by transaction 3 and all these, things there may be many, many, many rollbacks, just because one transaction abort and one transaction rollbacks. So, this is the series of cascading rollbacks. So, that problem may be there, even if a transaction is recoverable. So, now, just to highlight the point a little bit better and just given an example, suppose this is the schedule is r 1 a, then there is a w 1 a, then there is a r 2 a, then there is a w 2 a, then there is a r 3 a, then there is a w 3 a, etcetera.

Now, as we send that the, because to make it recoverable, it should happen that r 1 should first commit, then r 2, then r 3, but suppose r 1 somehow besides to abort, there is an problem. Now, if sees r 1 aborts, r 2 must also abort, that means this should abort, because this is aborted and r 3 must also abort, so that is the thing. So, that means, a 1 leads to aborting of a 2 and then leads to aborting of a 3. So, by the way a 1 stands for abortion and c it for commit.

So, commit and abort a c and i, so that is the series of rollback. So, that is the problem is recoverable schedules is that, even though it is fine in the sense of correctness, but it may lead to a lot of work being undone. So, there may be a lot of time to recover, because lots of aborts are done and lots of time it is takes for the schedule for the database to recover.

(Refer Slide Time: 08:02)



Cascadeless schedule

no cascading rollbacks.

1. T_i reads x written by T_j

$\Rightarrow$ 2. T_j commits before T_i reads

$S: r_1(a)\ w_1(a)\ r_2(a)\ \underline{r_1(b)}\ a_1\ \overset{a_2}{\cancel{c_2}}$ ✗.

$S: r_1(a)\ w_1(a)\ r_1(b)\ \overset{a_1}{\cancel{c_1}}\ r_2(a)\ c_2$ ✓

no completed transaction rolls back

So, the next concept that comes in the space is the concept of cascadeless schedule. So, to avoid this problem of cascading rollbacks, we defined what is next called a cascadeless schedule. So, essentially cascading rollbacks are eliminated, so there are no cascading rollbacks. If a schedule is cascadeless; that means, that there should not be any cascading rollbacks and a more formal definition is the following is that, if transaction T i reads x, which is written previously by transaction T j.

Then, it must happen that transaction T j commits before T i even reads, so note the definition, note the different of the definitions from the recoverable schedule, in the recoverable schedule, the first condition must the same. So, if T i reads something, which is written by T j, T j must commit before T i commits that first was the recoverable schedule.

In this new cascade less schedule, T j must commit even before T i has rate. Now, the intuition should be clear, why this is being made. So, intuition is the following is that, if T j commits before the read is being done. So, if T j is successful; that means, the read is correct, there is no problem with the reading. If; however, that T j is wrong and it has abort it, it can be argue that the T i has not wasted in a operation, T i has not even rate from T j.

So, T i by itself does not need to abort, because it is not even done anything, which is dependent on T j. So, that is why it is avoids the cascading rollback of T i, if T j rolls back and that is the definition of cascadeless schedule. So, here is the example, once more the same example we will follow as we did. So, it is r 1 a, w 1 a, then there is an r 2 a and r 1 b.

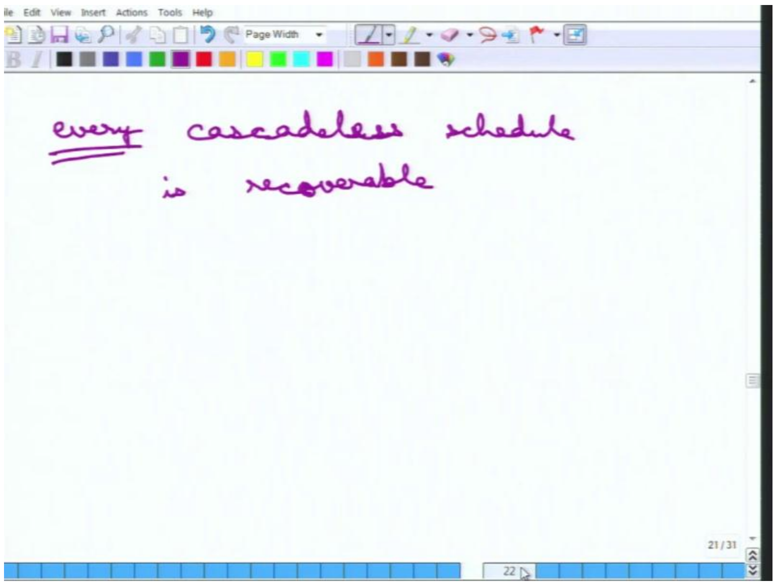
Now, the original problem was if this abort happens, then this commits should also be changed to abort, this is why this was recoverable, but not cascadeless. Now, to make it cascadeless, what needs to be done is the following is the r 1 a, w 1 a, then it cannot be that r 2 a does before a 1 does. So, transaction 1 must commit, but for transaction 1 to commit, it must complete this operation.

So, r 1 b, then transaction 1 commits and then, transaction 2 starts doing whatever it must to do, because this read depend on this, to this commit must happen before this reads happens. So, now, is the, what is the good thing about this is that, even if this abort, then of course, transaction 2 has not done anything at all. So, it just simply not started, it does not need to abort if the question of aborting or commit does not erase for t 2.

Because, it has not started anything at all, so this is not the first example, this is not cascadeless, but recoverable, but this cascadeless and recoverable. So, essentially the idea is that, no completed transaction needs to rollback, because t 2 has not completed. So, it does not need to rollback, so no completed transaction rolls back; that is the effect of this cascadeless schedules.

So, that is the definition the property of cascadeless schedules, now one thing is that there is a connection between cascadeless recoverable just like a view serializability and conflict serializability. There is again a connection between cascadeless and recoverable. As we saw in the example that the first example was recoverable, but not cascadeless and the second example was cascadeless and if you are noticed, it is also recoverable.

(Refer Slide Time: 12:02)

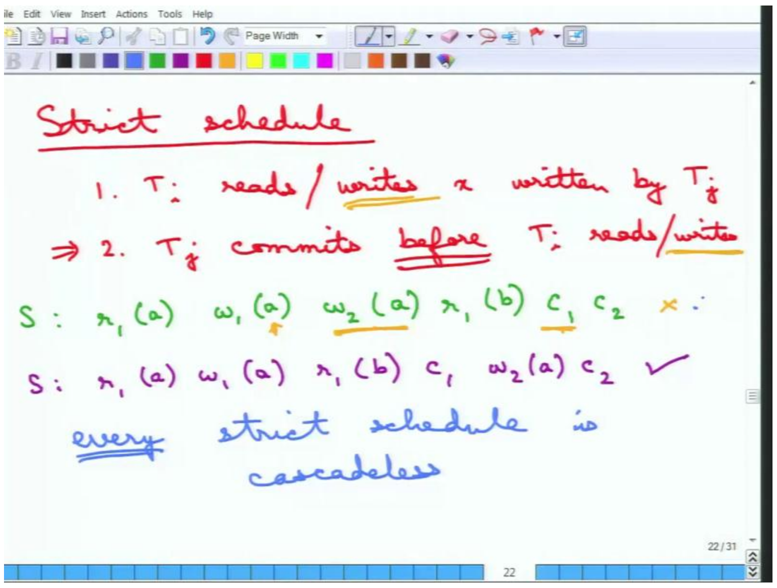

every cascadeless schedule is recoverable

Now, this is not an accident, so we can put this down as actually in the following manner is that every cascadeless schedule is necessarily is recoverable. Now, this is again, this can be shown in two ways from the intuition we can see plus also it can be actually proof by easily from the definition. Now, the definition of the cascadeless says is that T i does not read before T j commits; that means, that T i cannot commit of course, because T i is a reading operation, the commit is the last operation of a transaction.

So, if T i after T j commits, then of course, means that T i commits after T j commits, so that means, every cascadeless schedule is necessarily recoverable, but not the other way round and we saw example, why it is not the other way round. So, that is fine up to this cascadeless schedule, now this all seems to be correct, but there is still a little bit problem that is left, even after cascadeless schedule is that the problem of writes remain.

So, the writes, the problem of writes remains in the sense is that a later transaction. So, all the cascadeless schedule etcetera says that transaction T i reads, what has been written by T j. Now, which means that nothing is said about the writing of transaction T j, suppose the same data item is written by T j, it does not read what T i writes, but it simply writes the same thing. And then, it tries to commit before the commit of T i happens, now that is also of course wrong, because it should be writing after words not earlier.

(Refer Slide Time: 13:44)



Strict schedule

1. T_i reads / writes x written by T_j

$\Rightarrow$ 2. T_j commits before T_i reads/writes

S : $r_1(a)$ $w_1(a)$ $w_2(a)$ $r_1(b)$ c_1 c_2 $\times$

S : $r_1(a)$ $w_1(a)$ $r_1(b)$ c_1 $w_2(a)$ c_2 ✓

every strict schedule is cascadeless

So, the problem of writes is not handled not talked about at all by recoverable and cascadeless schedule so their mixed. Definition is that of strict schedule, where it tries to handle these situation with the writes. So, let us first look at definition of this, so essentially it is kind of a same thing, if T i reads or writes, this is the change here T i reads or writes x, which is previously written by T j.

Then, it must happen that T j commits before any of this read or write operation is done. So, before T i reads or writes, so it is not just about reads also about writes. So, it enhance as the definition of cascadeless to include the writes, so that is why, it is something more than cascadeless. Now, let us go to the example, suppose r 1 a, then w 1 a, then there is a w 2 a, then r 1 b, now essentially what it says is that, this is commit by c 1 and then commit by c 2 fine up to this part is fine.

Now, this is not strict, because this writes on the same data item x is happening before c 1 commits. So, this is not strict, although this is cascadeless, but this is not strict. So, this job need should have happen before c 1, now to make it strict, but the following thing needs to be done. So, this r 1 a, w 1 a, then of course r 1 b, then commit and then, this w 2 a and c 2, so this makes it strict.

Now, the question is the connection between strict and cascadeless, again it can be very easily seen that every strict schedule is cascadeless. But, of course, not wise versa, the reason is, so this is the example to show that this is cascadeless, but not strict, but otherwise it can be again very easily understood from the definition. So, strict argues when T i reads the writes from T j and it should the commit must happen before reads or writes.

So, if you just read, of course, the reads or write covers that reads, so that means, that every a strict schedule is of course maintains, what the cascadeless schedule it tries to say. So, it is of course, cascadeless plus not the other way round, because is an example the first example actually shows that. Now, all that things to be fine, but the question is why is strict important, I mean why do we bothered about this writing of a and before what happens, what is the problem with this schedule.

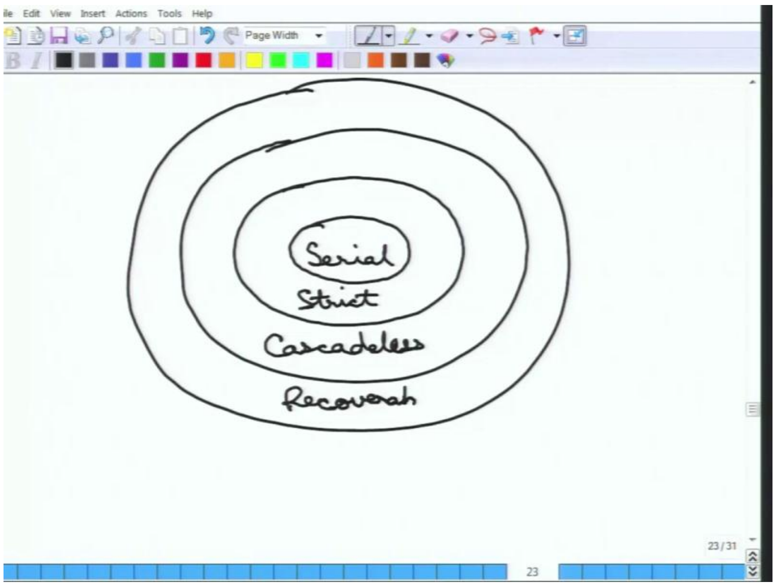
I mean, this we are saying that, this is not strict, what is the problem, the problem is the same as the earlier one is that. So, this the problem is the same as what we have been seen in the view serializability is that w 2 a essentially of blind write Now, which mean that w 2 a maybe may have rates some other value and if transaction 1 by itself aborts, it is may not be correct the transaction 2 simply blind writes and then, successfully goes to successfully commits, it should not be happening.

So, again it is a completed transaction that is allowed before it depends on some other transaction. So, essentially they write after write conflict is not taken care of, it take care of the write after write conflict, it must happen that the next write w 2 a must wait for the w 1 a to finish successfully, which means must wait for transaction 1 to commit. That is the big thing, so that is why, this is strict.

Now, just to complete the story, so by the way just one very interesting thing is to say that, every serial schedule, so what is the serial schedule, just to refreshing maybe it is a one transaction happens then completely commits and then, the next transaction starts and then, completely commits, then the third transaction. So, every serial schedule in which this follows it is of course, strict.

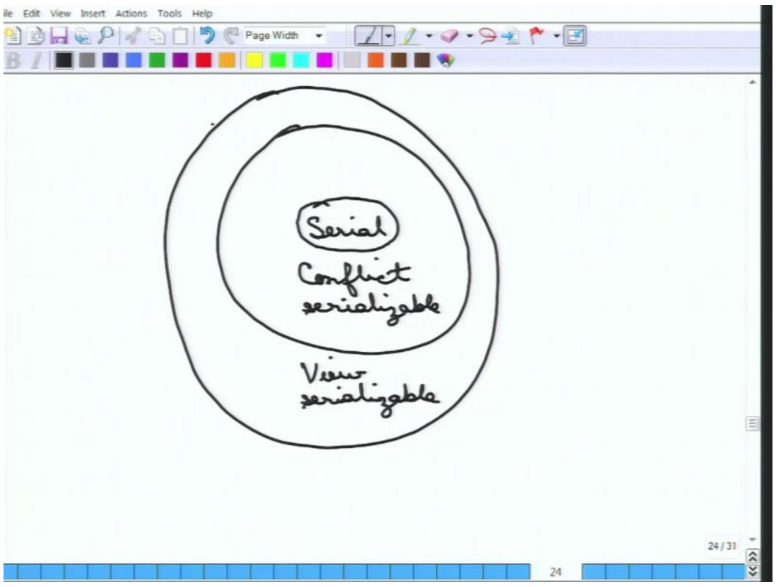
Because, a before the transaction 2 does anything, whether it is read writing the same data item etcetera, the transaction 1 has commit. So, it is of course strict, which means it is also cascadeless and recoverable, so here is the nice Venn diagram to think about.

(Refer Slide Time: 18:24)



So, there are these serial transactions, this is serial, now if it is serial, it can be also thought of as, so there are then these things, which are strict. So, serial transactions are strict, but there are other schedule, which also strict, but not serial. Then, there are cascadeless schedules, which I mean every strict is of course, cascadeless, but not the other way round and then finally, there are recoverable schedules. So, that is seems to be a nice Venn diagram.

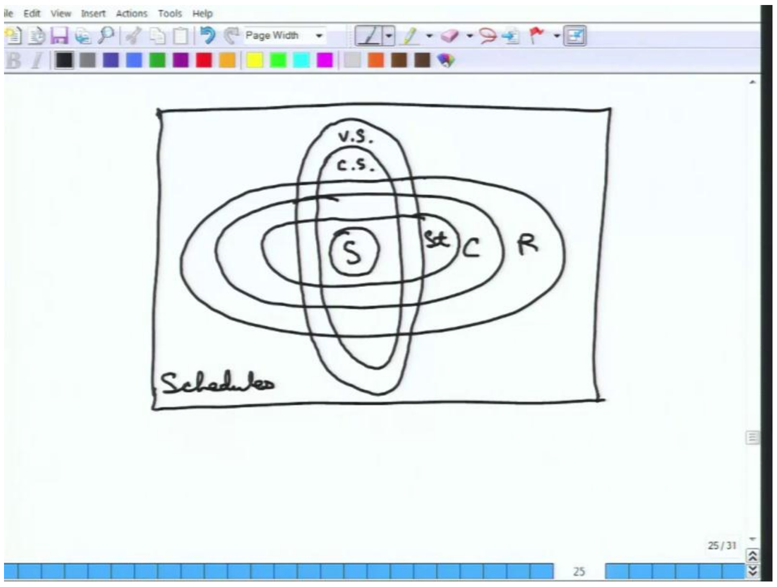
(Refer Slide Time: 19:09)



Now, if we repeat the Venn diagram for if we go back to the concept of serializability and we can write, we can produce the similar Venn diagram. So, if it is serial; that means, that it is also conflict serializable, but of course, there are other conflict serializable, which are not strict and every conflict serializable schedule is also view serializable, but not the other way round.

So, we can again thing of some another Venn diagram like this, now the question is the two Venn diagrams, let me kind of very quickly abbreviate them and write on what I am trying to say. So, suppose this serial schedule is S, now that is your S.

(Refer Slide Time: 19:51)



So, what is the connection, essentially what I am trying to say is what is a connection between recoverability and serializability and here is the Venn diagram that it shows is that suppose, so this is a let us say conflict serializable. Then, let us say this is a view serializable, then there can be schedules which a strict. So, this is strict what does it mean that they are schedules which as strict, but neither view serializable nor of course, conflict serializable. Then, there can be strict schedule, which has view serializable, but not conflict realizable.

Then, there are strict schedule which as view serializable, but not conflict serializable, then there are strict schedule that the view conflict serializable and that means, of course, view serializable, but strict and conflict serializable. And strict and then, there can be this cascadeless and then, finally there can be recoverable. So, operating all possible combinations are there and of course, just to complete this, this is the entire space of schedules and there are some schedules, which as not at all view serializable and not recoverable.

So, but the interesting part is here is that serializability and recoverability apparently can take any parts of there can be schedules which so the recoverability is independent of the serializability. So, whether the schedule is recoverable or not says nothing out whether the schedule is serializable or not and wise versa. So, apparently you can fine example for all of them and that is the nice thought example or whatever you can do this a nice homework for you to do about.

So, thing about can we get examples of schedules, which following one of these things and the other. So, all the portions of the Venn diagram can you get example of follow up them. So, that completes the module and schedules and we will next start on concurrency control protocols in the next module.