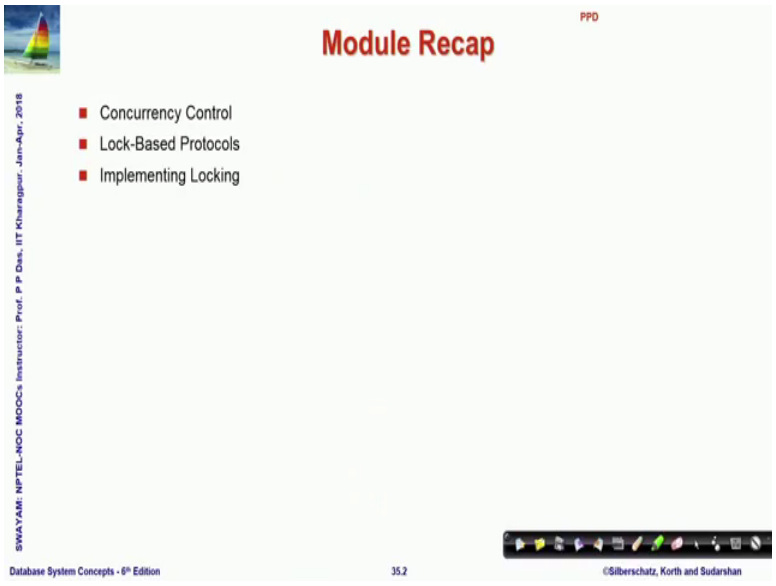
Database Management System
Prof. Partha Pratim Das
Department of Computer Science & Engineering
Indian Institute of Technology, Kharagpur

Lecture – 35
Concurrency Control/2

Welcome to module 35 of Database Management Systems. We have been discussing about concurrency control, this is a second and concluding module on that.

(Refer Slide Time: 00:29)



Module Recap

- Concurrency Control
- Lock-Based Protocols
- Implementing Locking

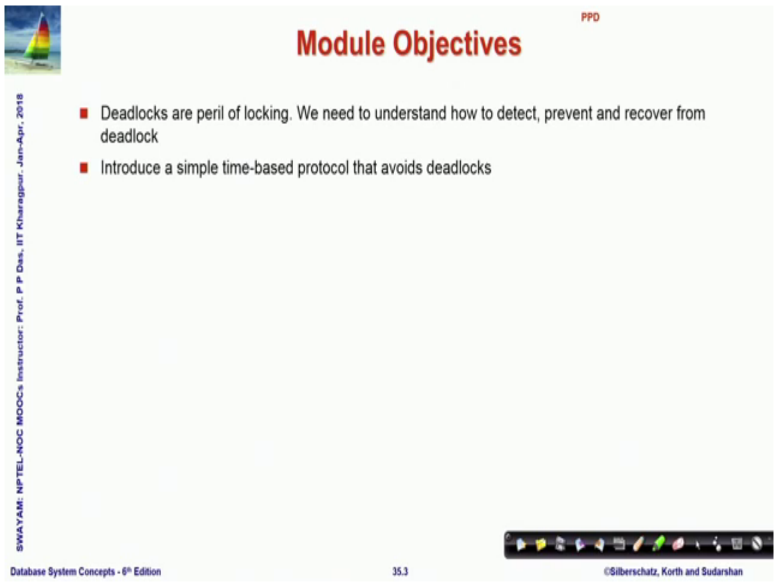
PPD

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.2 ©Silberschatz, Korth and Sudarshan

So, in the last module, we have talked about the basic issues in concurrency control; and particularly talked about log based protocol and how to implement locking in very simple terms.

(Refer Slide Time: 00:39)


PPD

Module Objectives

- Deadlocks are peril of locking. We need to understand how to detect, prevent and recover from deadlock
- Introduce a simple time-based protocol that avoids deadlocks

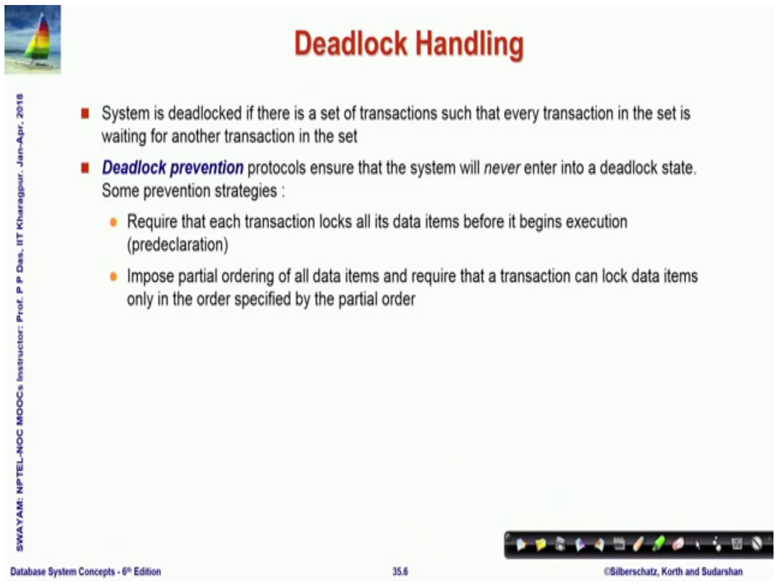
SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.3 ©Silberschatz, Korth and Sudarshan

As we have seen that deadlocks of the perils of locking I mean we cannot do without locking and certainly if we lock then deadlocks are inevitable almost to happen. So, here first we try to understand how since dead locks are inevitable.

So, there has to be mechanisms to detect deadlocks and recover from them. And also we would like to look at if it is possible to create strategies which can prevent deadlock from happening at all. And so after having studied that we would like to understand take a look into a simple time-based protocol that can avoid deadlock.

(Refer Slide Time: 01:27)


Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration)
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.6 ©Silberschatz, Korth and Sudarshan

So, deadlock handling. So, system is deadlock if there is the again just to recap the simple idea is if there is a set of instructions such that every transaction in the set is waiting for another transaction in the set and therefore none of them can actually proceed. So, deadlock prevention protocol ensures that the system will never enter into the deadlock state.

So, the question is can we make some strategy. So, why are we getting into the deadlock, because transactions are making requests for different locks and those are granted. And then some more requests come and we come to a state where a is waiting for b, b is waiting for c, c is waiting for a kind of a situation and we get into a deadlock.

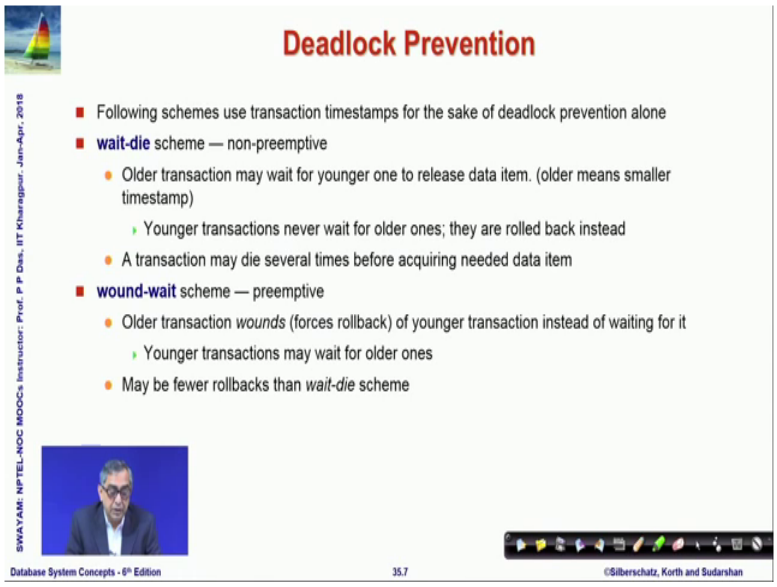
So, can we have strategies so that the requests and releases are done in a way, so that the deadlock will not happen at all. So, I mean fortunately such number of such strategies exist. For example, one strategy which is called a predeclaration which required that each transaction locks all debt items before it begins its execution that can be shown that that ensures that you will never have deadlock because where in very simple terms you will not be able to start before you have got all the locks.

And once you have got all the locks naturally you have every access to all possible data items and therefore, you will be able to proceed. Naturally, the flip side of this is this will delay the beginning of the transactions to a great extent in many cases, and particularly will bring down the level of concurrency that you can have.

The other which is smarter is what it does is imposes a kind of partial ordering of all the data items that a transaction and all the data items that exist. And it requires that the transaction can lock the items in only in that specific order. So, the important thing here is a partial order among the data items.

And the fact that you locked data items in that order that is specified by the partial order, you cannot lock out of order. And if you can do that then it can be shown that the deadlock will get prevented. We cannot we do not have time to go into the details of how that works, but I just want you to know that such strategies of prevention exist.

(Refer Slide Time: 03:47)



Deadlock Prevention

- Following schemes use transaction timestamps for the sake of deadlock prevention alone
- **wait-die** scheme — non-preemptive
  - Older transaction may wait for younger one to release data item. (older means smaller timestamp)
    - Younger transactions never wait for older ones; they are rolled back instead
  - A transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it
    - Younger transactions may wait for older ones
  - May be fewer rollbacks than *wait-die* scheme

So, the other possible prevention schemes that we will we would like to look at little bit more depth is the fact that I can use timestamps for the transaction. And use those timestamps that is which will tell me which is an earlier transaction in which, a later transaction for preventing deadlock and several strategies for that exist. We will discuss two of them. First is what is known as wait and die with scheme, which is non-preemptive. Non-preemptive means that well in this no one preempts anyone else.

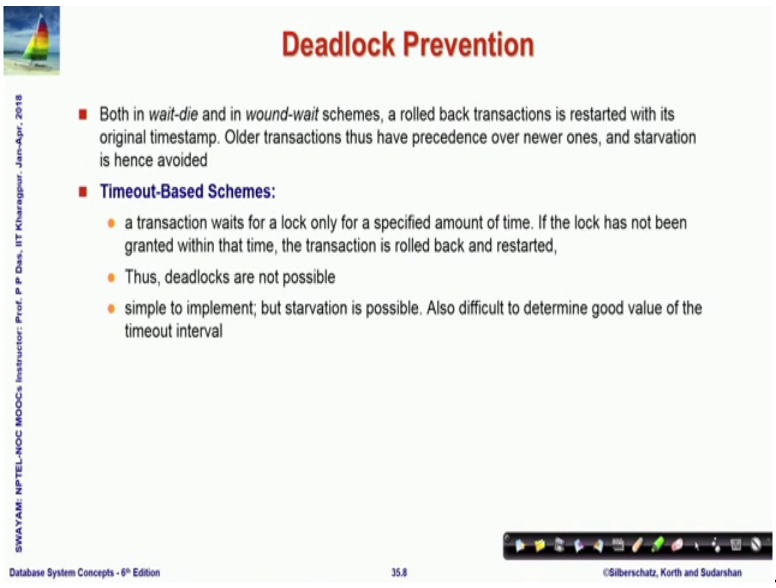
So, what do you do in wait and die, the older transactions because you assume that every transaction has a timestamp. So, smaller the timestamp, older is a transaction. So, older transaction may wait for the younger one to release the item. So, if two transactions are conflicting then the older one will wait; and the younger transaction will never wait for the older one, they are rolled back instead.

So, if there is a conflict, then the younger one in that will always roll back, and the older one will wait. So, a transaction may die several times before acquiring the needed data item kind of starvation may happen, but certainly there will not be a deadlock. Because my a waiting on b, and b waiting on a, cannot happen because out of a and b one must be older has to be older, and that only will wait, the other one will abort, abort and roll back on.

The other is a preemptive scheme where which is called wound and wait scheme, where the older transaction wounds up or forces a rollback of the younger transaction instead of waiting for it that is why this is preemptive. So, the older transaction is preempting the young that transaction to continue to wait and forces it to roll back to abort and that, but the younger transaction may wait for the older one.

So, by doing this preemptive one also it is possible to have a fewer roll backs than the other scheme. So, it is a preemptive scheme, but it the advantages it might allow you fewer roll backs to happen. And with these two kind of timestamp based schemes it is possible to actually prevent deadlocks and for that reason these kind of schemes are often preferred in many context.

(Refer Slide Time: 06:21)



Deadlock Prevention

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided
- **Timeout-Based Schemes:**
  - a transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time, the transaction is rolled back and restarted,
  - Thus, deadlocks are not possible
  - simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.8 ©Silberschatz, Korth and Sudarshan

So, both in wait and die, and wound and wait scheme, the rollback transaction is restarted with its original timestamp. This is a very very important point to note. When you restart, so your rollback so you have to restart that transaction you restart the transaction you do not put the timestamp of when it is being restarted, you put the timestamp of its original time; The time when it was started and had to be aborted and rollback.

So, the older transactions have precedence over the newer ones and that starvation will get avoided.

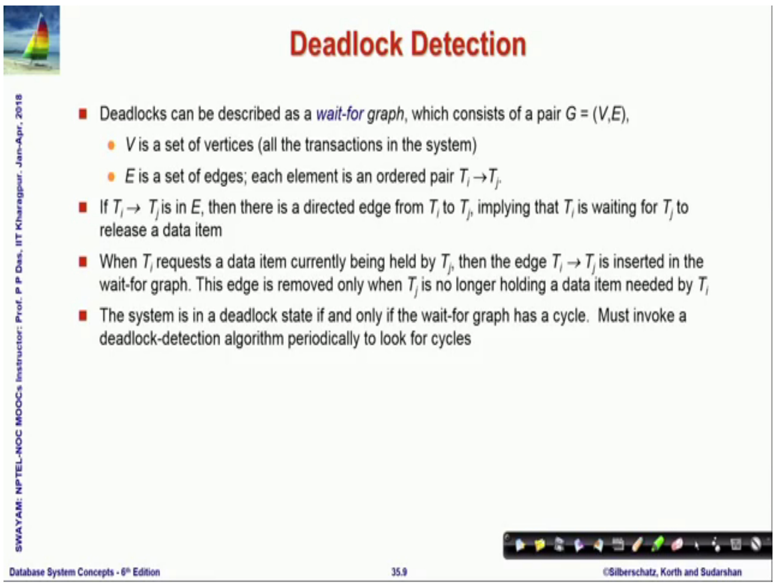
So, now what becomes you are you are actually a new candidate because you have been rolled back in and started again, but you carry your older timestamp. So, your precedence has gone higher because in wait and die, and wound and wait in both actually the older one has a precedence.

So, by carrying your older timestamp, you inherently bring in a higher precedence in the system. And in this way there is a precedence based ordering that will naturally always happen. So, this will not only avoid deadlock, but this will also ensure that starvation is avoided; So, very simple and nice scheme.

So, in this you usually have time out basically my transaction waits for a lock only for a specified amount of time. If the lock has not been granted within that time the transaction is rolled back and restarted, and therefore, the deadlock is not possible. It is simple to implement, but starvation can happen in the timeout based scheme. And it is also difficult to determine what is a good time interval to wait.

If you wait too short, then you will spend a lot of time in the in the rollback and restart. If you wait for too long, then your throughput will go down because several transactions are basically waiting on logs. So, theoretically it does avoid deadlock, but in terms of starvation and in terms of the practicality, this there are critical things to decide on this.

(Refer Slide Time: 08:24)



Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.9 ©Silberschatz, Korth and Sudarshan

The second issue in terms of deadlock that we must be able to answer is well hundreds of transactions are going on in the system. Now, how do you know that a deadlock has happened? Because if a deadlock has happened and if you are not using a preventive scheme to ensure that the deadlock will not will never happen theoretical proof; If you are allowing say two phases are locking kind of protocol where deadlocks can happen then you must know what the must be able to detect that a deadlock has happened and then take care of it to rollback the transaction.

So, for doing this, we again create a graph, which is wait for graph which is very similar to the precedence graph we saw earlier which the nodes are the transactions and the edges are ordered pair of transactions. So, what do you put an edge from T i to T j, you put this edge in what it means is T i is waiting for T j. So, if we have a a conflict, then certainly one transaction is holding the lock and other is other has requested for that lock.

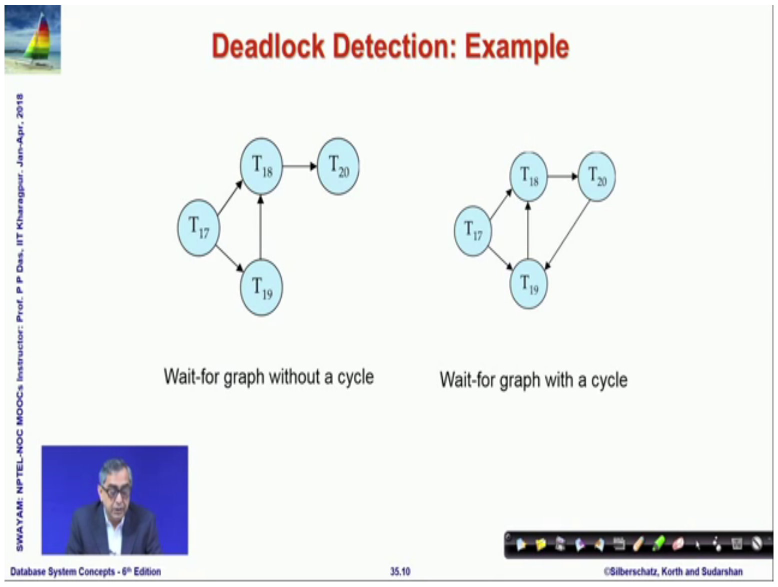
So, what you do you put an edge for from the one that is waiting for the lock to the one that is already holding the lock for the release of the lock, and in this way the graph gets built up. So, naturally when T i requested it item currently being held by T j, then the edge T i T j is inserted in the in this graph. And when the release happens then this edge is removed because T j is no more holding the item that T i had actually required.

So, this is how this is kind of a dynamic graph the wait for graph is a kind of dynamic graph which will regularly keep getting updated. Now, naturally from the description of this graph, you can understand that a deadlock if a deadlock has to happen then this graph must have a cycle. So, if at any instant the graph has a cycle then there is a deadlock; otherwise the graph will grow and shrink grow and shrink it will keep on happening that way.

So, it is important to ensure that this graph remains a cyclic which now this is dynamically happening hundreds of transactions, transactions are getting created, they are getting committed, aborted, they are requesting logs they are releasing logs and so on. So, how do you ensure that the graph at every stages is remaining a cyclic or a cycle has happened and therefore, a deadlock is actually happening.

So, what you will need to do is periodically run another process which invokes the deadlock-detection in the graph that is it looks for the cycles, and the cycle is there, then you have to do some strategy to roll back about one of the transactions, and break the cycle and then so that the other transaction can proceed.

(Refer Slide Time: 11:29)



Deadlock Detection: Example

$$\begin{tikzcd} & T_{18} \arrow[r] & T_{20} \\ T_{17} \arrow[ur] \arrow[dr] & & \\ & T_{19} \arrow[uu] & \end{tikzcd}$$

Wait-for graph without a cycle

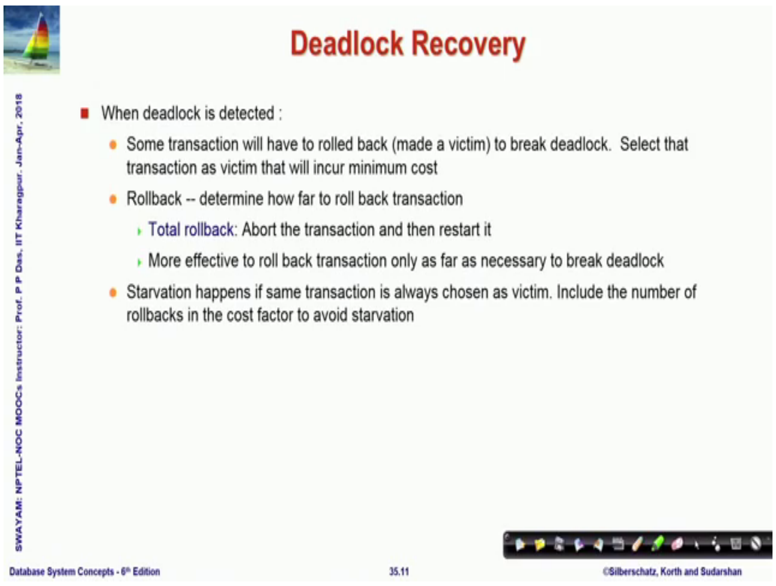
$$\begin{tikzcd} & T_{18} \arrow[r] & T_{20} \arrow[dl] \\ T_{17} \arrow[ur] \arrow[dr] & & \\ & T_{19} \arrow[uu] & \end{tikzcd}$$

Wait-for graph with a cycle

So, these are examples of the wait for graph. For example, here on the left as you see if you if I if I may point out in the left, if we see that T 17 is waiting for T 18 and T 19, T 18 is waiting for T 20. So, eventually and T 20 is waiting for none. So, at some point of time T 20 will be done and when that is done then T 18 would be able to proceed. And if T 18 is able to proceed then T 19 would be able to proceed, and then T 17 would be able to proceed.

So, there is no possibility of a deadlock, whereas in here if you look in the graph on right, then you can see that between these three they are waiting on each other. So, no matter how long you wait this will this deadlock will never be broken, and the deadlock-detection system has to detect this cycling and decide to abort one of these transactions and so that the rest of the transactions can progress. So, this is a sample deadlock-detection mechanism.

(Refer Slide Time: 12:44)



Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it
    - More effective to roll back transaction only as far as necessary to break deadlock
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.11 ©Silberschatz, Korth and Sudarshan

So, when the deadlock is detected there has to be a recovery. So, trump transactions will have to be rolled back to break the deadlock. And so there is a there is a strategy required to select which transaction must rollback; naturally that should be done based on the minimum cost that is you do not want because if you roll back then the recomputation naturally because you have to restart and do that transaction again.

So, you have to in terms of rollback, you have to determine how far to rollback that transaction. There can be a total one, so that you roll back the whole transaction abort and then restart it or you can roll back to a previous point, we discussed notions of safe point in the transaction program.

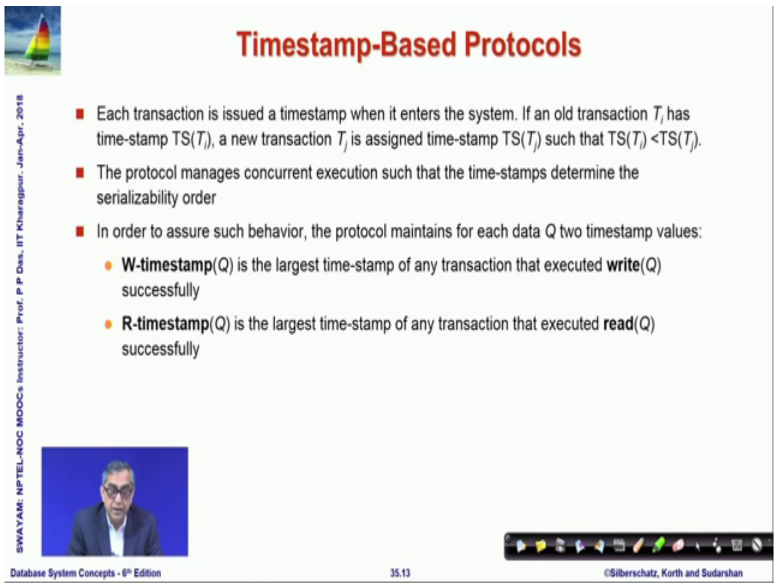
And so it is be more effective to roll back transaction only as far as necessary to break the deadlock, you may not need to roll back everything. Maybe this is this transaction is participating in the deadlock, because it is holding some exclusive lock which it took three instructions before. But before that it has done 300 instructions it is not necessary to rollback the whole of the 300 instructions, you can just roll back up to the point where it took that exclusive lock which is creating the problem.

So, that those are some of the strategies which can improve the throughput and minimize the possibility of starvation.

Starvation will again happen if the same transaction is chosen as a victim to be rolled back every time, which the possibility exists. And so the number of roll backs is also usually kept as a cost factor. So, when you roll back a transaction, you also keep a number saying that how many times this transaction has been rolled back.

So, higher that cost becomes then you would like to avoid doing the rollback for that transaction because so that it does not wait infinitely in terms of (Refer Time: 14:45) starvation. So, these are some of the simple strategies that roll back the deadlock recovery can be done.

(Refer Slide Time: 15:04)



Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order
- In order to assure such behavior, the protocol maintains for each data Q two timestamp values:
  - **W-timestamp**(Q) is the largest time-stamp of any transaction that executed **write**(Q) successfully
  - **R-timestamp**(Q) is the largest time-stamp of any transaction that executed **read**(Q) successfully

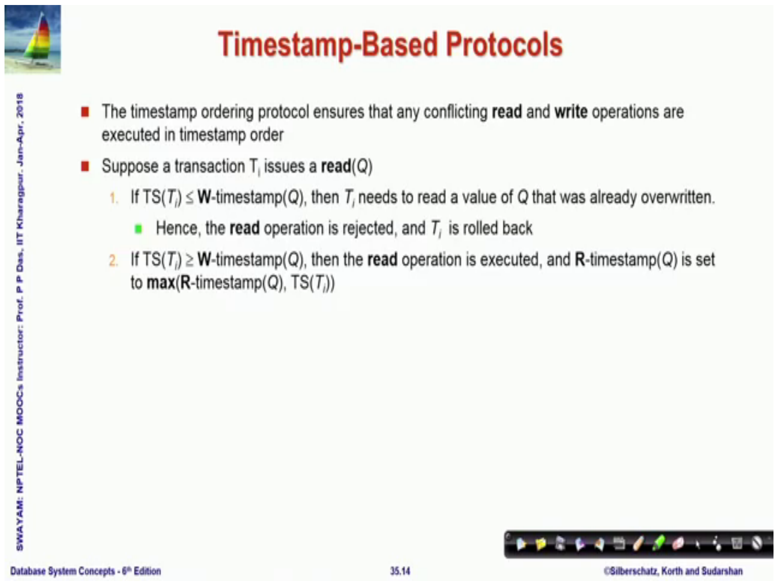
SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.13 ©Silberschatz, Korth and Sudarshan

So, having talked about the prevention detection and recovery from deadlocks let us quickly look at a simple time-based protocol in contrast to the two phase locking protocol we had earlier. This protocol does not lead to deadlock. So, what you do in here is each transaction is issued a timestamp when it enters the system. So, hold at the transaction, less is the value of the timestamp so that is a simple. So, time goes in the increasing order.

Now, the protocol manages a concurrent execution such that timestamps determine they themselves will determine the serializability order, they will determine in which order the transaction should occur. And for that for each data item two timestamp values are maintained; one is a right time-stamp on the data item queue, another is a read timestamp. So, this is the latest read write and read times for the data item. So, w timestamp Q is the largest time stem of any transaction that executed a write Q successfully. So, naturally what it means the largest timestamp means the latest write that has happened. Similarly, it keeps it latest read.

(Refer Slide Time: 16:15)



Timestamp-Based Protocols

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order
- Suppose a transaction T_i issues a **read**(Q)
  1. If TS(T_i) ≤ **W**-timestamp(Q), then T_i needs to read a value of Q that was already overwritten.
     - Hence, the **read** operation is rejected, and T_i is rolled back
  2. If TS(T_i) ≥ **W**-timestamp(Q), then the **read** operation is executed, and **R**-timestamp(Q) is set to **max**(**R**-timestamp(Q), TS(T_i))

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018
Database System Concepts - 6th Edition
35.14
©Silberschatz, Korth and Sudarshan

Now, using that you build up this protocol, so it is again looks only at conflicting read and write operations, and they are executed in timestamp order. So, let us suppose that let us consider the case of read. So, a transaction T i has issued a read.

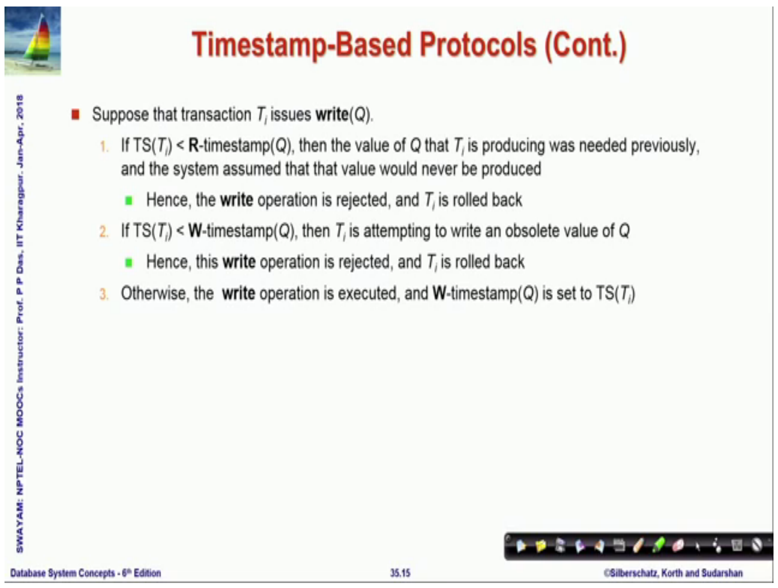
Now, if that the timestamp of T i is less than equal to W-timestamp Q which means that W times stamp Q is the latest write. And T s T i is a timestamp of the transaction. So, the transaction is older than the latest write. So, the transaction T i needs to read a value that was already overwritten because the latest write has happened after the transaction. So, this read operation can be rejected and T i will be rolled back.

If it is in contrast, if the timestamp of the transaction is greater than the latest right time W-timestamp Q then the read operation is executed and since we are doing a read operation. So, this becomes the latest read operation and therefore, the read timestamp R timestamp Q is set to the maximum of the current read timestamp and the timestamp of the transaction. Mind you here we the timestamp one confusion that may come to your mind is are we looking at the exact time when the read has happened or when the write has happened, no, we are all of this reasoning is happening with the timestamp of the transaction.

So, whenever it started. So, it is a older transaction and newer transition that we are reasoning with. So, when you update R-timestamp then you are the R-timestamp already has a value which is the timestamp of the latest transaction that has read that value and TS T i is the timestamp of the transaction that is read it now.

So, it is not always that since this read is the last read you will update this. So, by the sense of latest what I mean is the latest in the sense of the timestamp of the transaction that is reading it. So, you will compute that in terms of finding the maximum of the current read timestamp and the timestamp of this transaction.

(Refer Slide Time: 18:48)



Timestamp-Based Protocols (Cont.)

- Suppose that transaction T_i issues **write**(Q).
  1. If TS(T_i) < **R**-timestamp(Q), then the value of Q that T_i is producing was needed previously, and the system assumed that that value would never be produced
     - Hence, the **write** operation is rejected, and T_i is rolled back
  2. If TS(T_i) < **W**-timestamp(Q), then T_i is attempting to write an obsolete value of Q
     - Hence, this **write** operation is rejected, and T_i is rolled back
  3. Otherwise, the **write** operation is executed, and **W**-timestamp(Q) is set to TS(T_i)

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

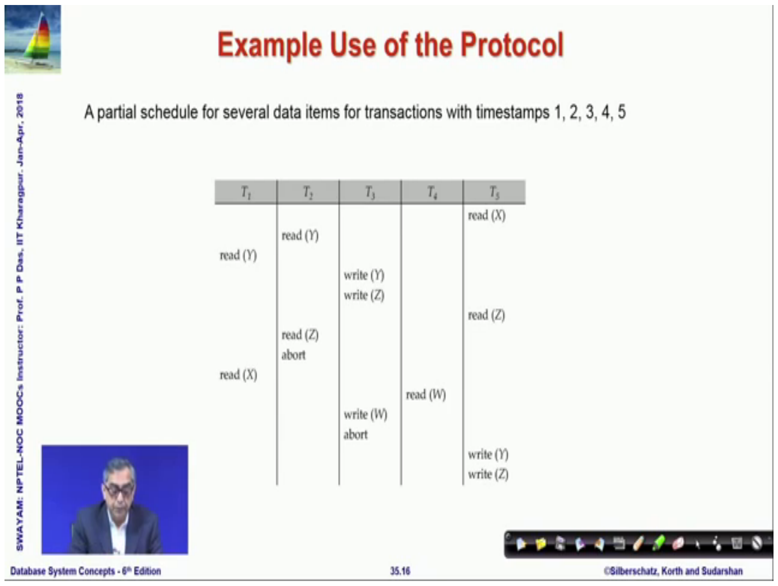
Database System Concepts - 6th Edition 35.15 ©Silberschatz, Korth and Sudarshan

Write is a little bit more complex, but we can reason in the same way. So, if T i issues a write Q. And if the timestamp of T i is less than R-timestamp that is if this transaction is it is less so it is older than the read timestamp that is it is older than the transaction that read Q last, the youngest transaction that read the value of Q.

So, then the value Q that T i is producing was needed earlier, it is trying to write, but already a newer transaction has used the value. And the system assumed that what the T i was supposed to write was not available was not produced, hence the write operation is rejected T i does not should not write this and T i will be rolled back.

Second case if the transaction T i has a timestamp which is less than the write timestamp. So, which means that this transaction is older than the transaction that has done the last write; So, T i is attempting to write an absolute value of Q, and hence this write operation is again rejected and T i is rollback. Otherwise, in other cases, the write operation is executed and the write timestamp will be set to the timestamp of this transaction which has written it. So, this is a very simple protocol for read and write.

(Refer Slide Time: 20:36)



Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

T_1	T_2	T_3	T_4	T_5
				read (X)
	read (Y)			
read (Y)				
		write (Y)		
		write (Z)		
				read (Z)
	read (Z)			
	abort			
read (X)				
			read (W)	
		write (W)		
		abort		
				write (Y)
				write (Z)

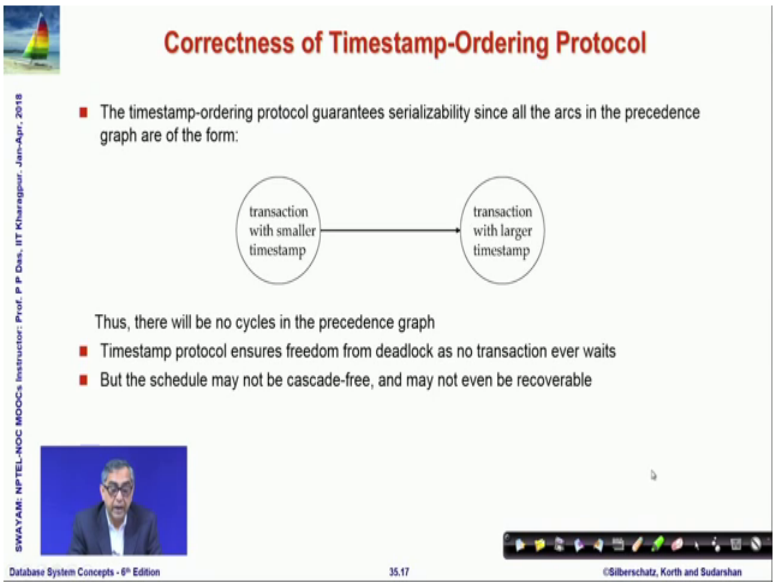
SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.16 ©Silberschatz, Korth and Sudarshan

And here is an example shown in terms of this protocol. For example, this wanted to do a read and this so this is the that this is a time where the transaction had started so and this was the write that. So, when this read is happening this is the so this is naturally this is at hold at transaction than T 3. So, this at this point, the right timestamp is that of T 3 and this is an older one, so it was trying to read that, so this was aborted.

Similarly as you see here if I clean and start again if we look at write W this is trying to do read and T 4, so read will this read has a timestamp which is of T 4 which is later than the timestamp of T 3. So, this gets aborted. So, you will need to spend a little bit of time to convince yourself that this will never actually ensure never actually lead to any deadlock, and it is a very effective serializable and simple strategy to ensure serializability while it avoids the deadlock.

(Refer Slide Time: 22:08)



Correctness of Timestamp-Ordering Protocol

- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

transaction with smaller timestamp
transaction with larger timestamp

Thus, there will be no cycles in the precedence graph

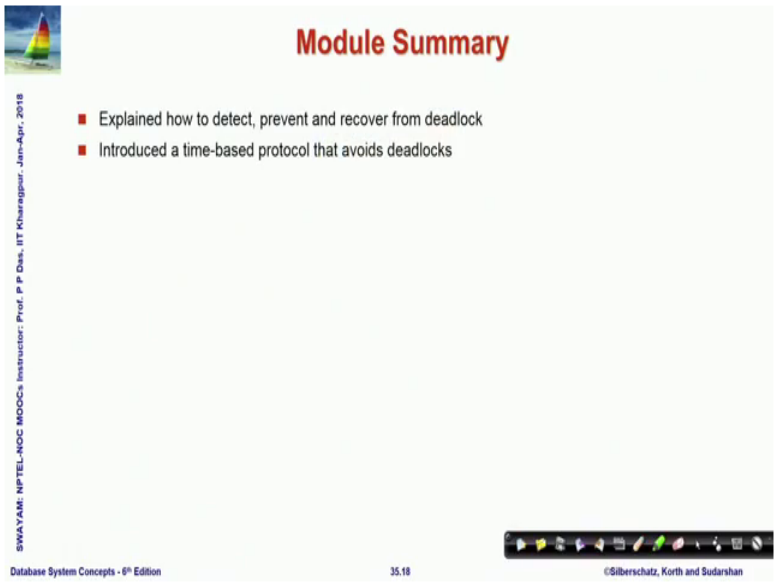
- Timestamp protocol ensures freedom from deadlock as no transaction ever waits
- But the schedule may not be cascade-free, and may not even be recoverable

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018
Database System Concepts - 6th Edition
35.17
©Silberschatz, Korth and Sudarshan

At all the timestamp ordering protocol itself guarantees the serializability, so the transaction with the smaller timestamp will lead to transaction with larger timestamp, because those are the more recent transaction. So, since the ordering is always in this manner, there cannot be any cycle in this precedence graph, because if they are recycle then naturally, somewhere you are you will be coming from newer to an older transaction which is not allowed in this protocol.

So, there cannot be a cycle and so this ensures that there cannot be deadlock in this time-based protocol, but the schedules that they produce they may not be cascade free. And actually examples can be shown that they may not even be recoverable there may be some irrecoverable schedules that get produced through this time-based protocol.

(Refer Slide Time: 23:10)



Module Summary

- Explained how to detect, prevent and recover from deadlock
- Introduced a time-based protocol that avoids deadlocks

SWAYAM: NPTEL-NOC MOOCs Instructor: Prof. P P Das, IIT Kharagpur. Jan-Apr, 2018

Database System Concepts - 6th Edition 35.18 ©Silberschatz, Korth and Sudarshan

So, to summarize we have tried to take a look into explaining what are the different ways to prevent deadlock; Some of the strategies and specifically we focused on the time-based strategy. So, there are some strategies which are based on the order of accesses the data items ordering of data items and so on. And we have specifically focused on time-based strategies.

And from that there are multiple time-based strategies which can prevent deadlock. And in case the deadlock has happened then we have discussed a simple wait for graph data structure and algorithm to be able to detect that the deadlock has happened. And if once this has been detected, we have talked about basic strategy to recover from that that is how do you decide what are the what is a good candidate victim transaction which should be rolled back, which should be aborted.

And on that study we have presented a simple time-based protocol which maintains the timestamp of transactions to decide the ordering in terms of read and write and deciding as to whether you should continue doing a read or write or you should abort the read and write attempt; And thereby ensuring that deadlocks do not happen in the system though there may be other problems in this in terms of having cascading rollback or having some irrecoverable schedules at times.