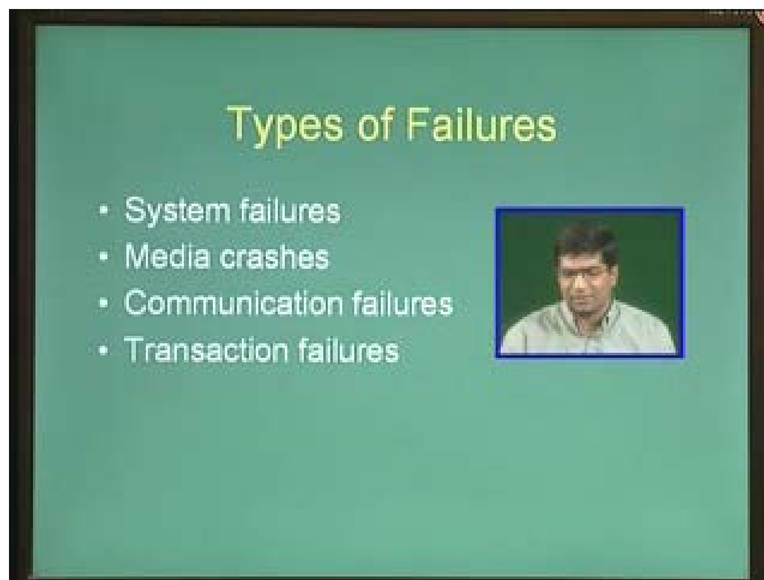
Database Management System
Dr.S.Srinath
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 28

Recovery Mechanisms II

Hello and welcome. In the previous session we started looking into recovery mechanisms in databases, especially we looked into the back ground of transactions and the idea of a transaction and how recovery should maintain consistency in terms of different transactions. That is it should not leave a transaction in a unatomic form that is when a database is consistent or either all transaction is performed completely or they have not been performed and it should not violate an integrity constraints and it should be serializable and so on.

Today's session we are going to look at some mechanisms for recovery. Specifically we are looking into what are called as log based recovery. As the name suggests log based recovery means that recovery mechanisms for the database are performed using transaction logs. That is whenever transactions happen certain elements of the transactions are logged into log files and using these log files, we can try to recover the database into a consistent state in case of any kinds of failures.

(Refer Slide Time: 2:37)



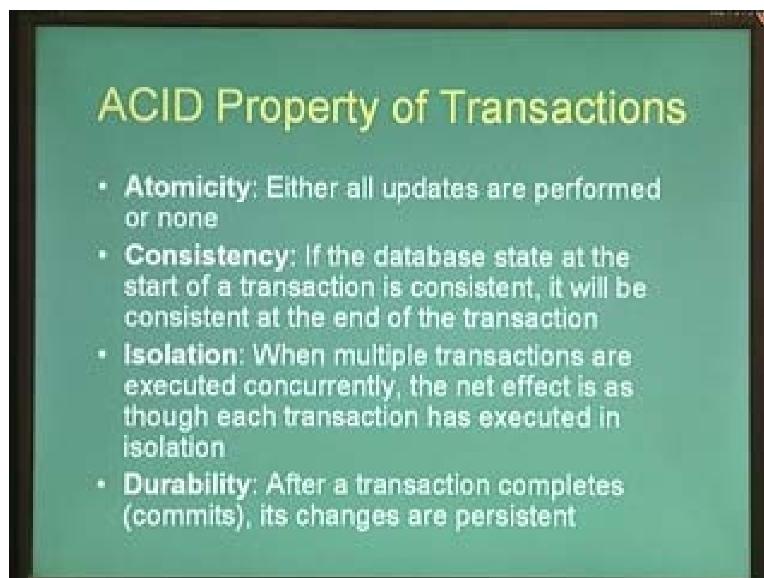
So let us briefly summarize whatever we have learnt about the transactional requirement of databases before we look into recovery mechanisms or log based recovery mechanisms. Firstly, why recovery or in what situations do we talk about recovery mechanisms? Recovery is pertinent in the phase of failures and given database system can be subject to different kinds of failures there could be system failures, the power

could just go off database, there could be media crashes that is the disc crash or something of that sort. There could be communication failures that is network has failed and transaction which was partially submitted or especially if you are having distributed databases, transactions which was started on other machines failed communication between the two machines failed and so on. And there are of course transaction failures, that is transaction could fail for a variety of reasons including the above kinds of failures that we are talking about. The transaction could fail because they violated integrity constraints, transaction could fail because they could not, there is no serializable schedule for the set of activities from these transactions or they could fail because whatever schedule that's being currently performed has led to a dead locker something of that sort.

So in many of these failures we need a recovery mechanisms. In the last case that is transaction failure usually its automatic that is the system is still functional the dbms, the database everything is still functional so it is just a matter of re submitting the appropriate transactions after waiting for a while and hoping that it succeeds this time rather than fail or if the transaction has violated an integrity constraints, it involves something like raising an exception or intimating the application program saying your transaction is wrong or I cant perform your transaction because for example your account doesn't have enough money to withdraw so much amount that you have asked something like that.

So leaving aside the last point here, in most of the other cases the dbms or the databases has crashed and it has to be booted up, it has to be brought up again. And once it is brought up again there is no guarantee that what ever data that's there in the database is consistent and it will be un safe to just start the database and have it running from where ever it is left off because any amount of data that was there in the volatile memory in the ram would have been lost and we don't know how we can set these things right.

(Refer Slide Time: 5:30)



So what are the properties of transactions that we have to assure when we are providing recovery mechanisms? The property of transactions as you know is called the acid property of transaction that is atomicity, consistency, isolation and durability properties. So let us briefly summarize what is meant by the acid properties and what do they require.

Atomicity means either all updates that are performed by the transactions are performed that is either all updates that are required by the transactions are performed or none of them are performed. We can’t have a transaction that has performed half of the updates that were made for it that is we cannot have a transaction that has debited my account in a wire transfer transaction and has not credited the other account and the money is lost. So it should be either all or none kind of operation.

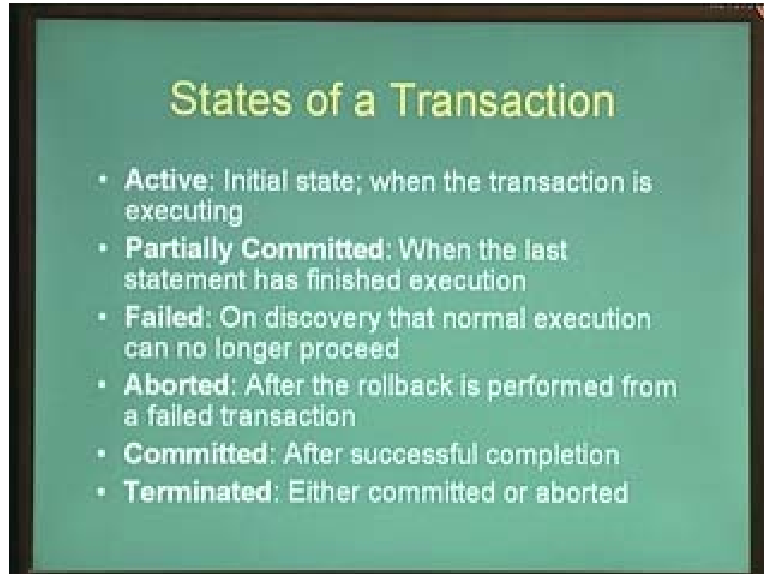
Similarly consistency requirements is that when a transaction has to finish or if a transaction has to successfully complete, it should not violate any integrity constraints of the database. That is given a consistent database a valid transaction should leave the database in another consistent state. It need not be the same state, it could be another state but it should be a consistent state. If it violates any kind of integrity constraints on the way then you have to roll back, you can’t complete the transactions and you cant leave it there as well because then we were violating the atomicity requirement of the transaction.

Isolation constraint that is the I in the acid property states that whenever there are multiple transactions that are happening on in a dbms, the net effect of all the transactional updates should be such that or should be equivalent to a schedule in which all updates of one transactions are performed before the first update of the next transaction is taken up. That is it should be as though the transactions have run in some serial order, it need not actually be run in serial order that’s what we saw in the previous session. Activities can be interleaved as long as this interleaving is safe. That is we saw notion of what is meant by safe that is the notion of conflict serializability that is we should be able to conflict or view serializability which we saw that is we should be … (Refer Slide Time: 08:13) into a serialize schedule without encountering any kinds of conflicts.

And the last property is that of durability. That is once a transaction commits it cannot be rolled back. The changes that are made after a transaction commits are durable, it is persistent and not only that the changes are made inside the database that is on to disk, the changes could also entile performing some kind of physical operation like we gave the example yesterday of dispensing money from an atm. that is if once a transaction for withdrawing money succeeds and it commits then the atm dispenses the required amount of money that was asked by the customer for withdrawal.

Now once the money is dispensed you cannot roll back the transaction. the problem … (Refer Slide Time: 09:13) if it is found that there was some error and the money shouldn’t have been dispensed, you have to look at solutions that go beyond the database systems, you cant ask the database to just roll back this transaction and leave it at that.

(Refer Slide Time: 10:31)



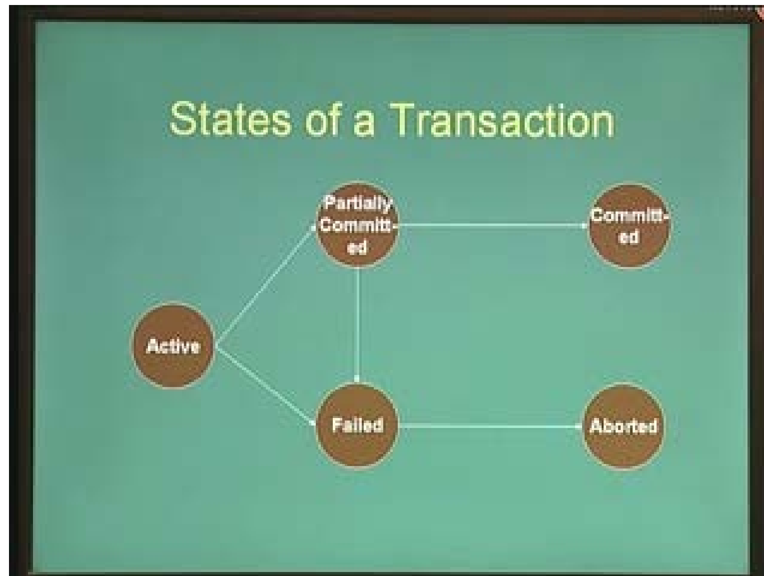
What are the different states in which transaction lies? The first state is the active state that is whenever a transaction becomes active and it is executing, it is said to be in an active state. Once a transaction has performed all its updates and it is ready to commit that is it has finished its executions and it is ready to commit then it is called a partially committed state. And once a transaction discovers that it cannot commit, mainly for example it has violated an integrity constraints or its schedule cannot be serialized and so on then it is said to be in a failed state. And once a transaction has rolled back from its failed state that is it has undone whatever it had done already then it is said to be in an aborted state. And if the transaction has committed successfully then it is in a committed state and either aborted or committed state is called a terminated state.

Now let us look at these states of a transactions in terms of recovery. That is what kinds of states require recovery of a transaction. now if a transaction is terminated, its either aborted or committed then we wouldn't have lost atomicity as a part of the transaction that is if the database crashes after a transaction has committed or aborted, it should be such that atleast the dbms should be designed such that these transactions should not be executed again. That is we should not submit this transaction again to the dbms.

For example if the user has requested for withdrawal of say 1000 rupees from an atm and the transaction has aborted or rather the transaction has committed and 1000 rupees has been dispensed from the atm and right after commit, the system crashes then the dbms should be designed such that whether or not this data is there on or has been updated on the dbms for a variety of reasons which will see shortly. Whether or not this data is updated on the dbms this transactions should not be run again. That is the user should not be given another set of 1000 rupees after the system comes back because the transaction has already run and the operation has already been performed whatever operation has been asked for.

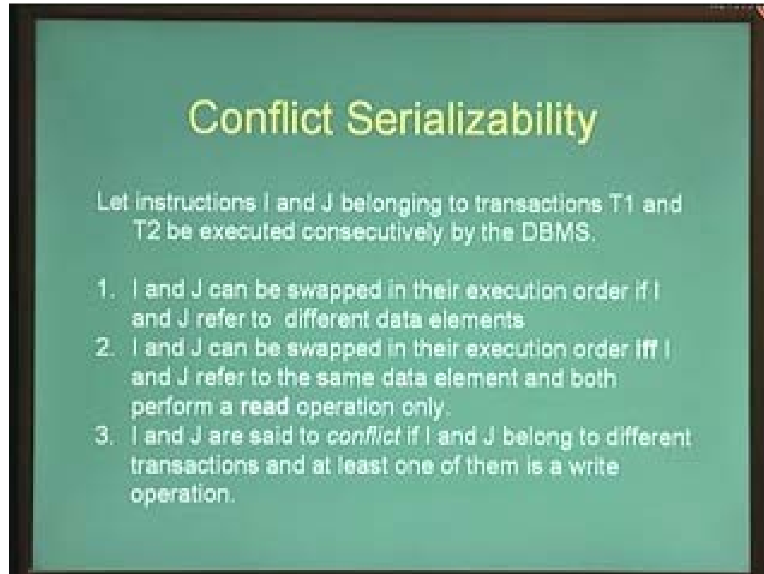
On the other hand if a failure occurs during let us say active or partially committed state then we may have to, in some cases undo whatever has been done by the transactions, whatever operations has been done by the transactions and then probably resubmit the transactions. That is re run the transaction once again from the start.

(Refer Slide Time: 13:54)



This slide shows the state diagram or state transition diagram for the different sets of a transactions. That is we start from the active state and the active state can go to either partially committed state or a fail state depending on whether all operation in the transactions have been executed successfully or whether there have been some problems. And even in a partially committed state, there is a chance of failure if the transaction finds out that it cannot commit. For example if the transaction is dependent on some other transaction to commit in case of and the other transaction rolls back and this transaction is subjected to a cascading roll back. So in that case even if all the operations have been… (Refer Slide Time: 13:27) there is still a chance of failure even from the partially committed state. And if nothing goes wrong then we can go ahead and go to the committed state or once we reached a failed state then the transaction goes in to the aborted state. That is it rolls back whatever been done, so it undoes whatever operations has been done and it goes back into an aborted state.

(Refer Slide Time: 16:14)

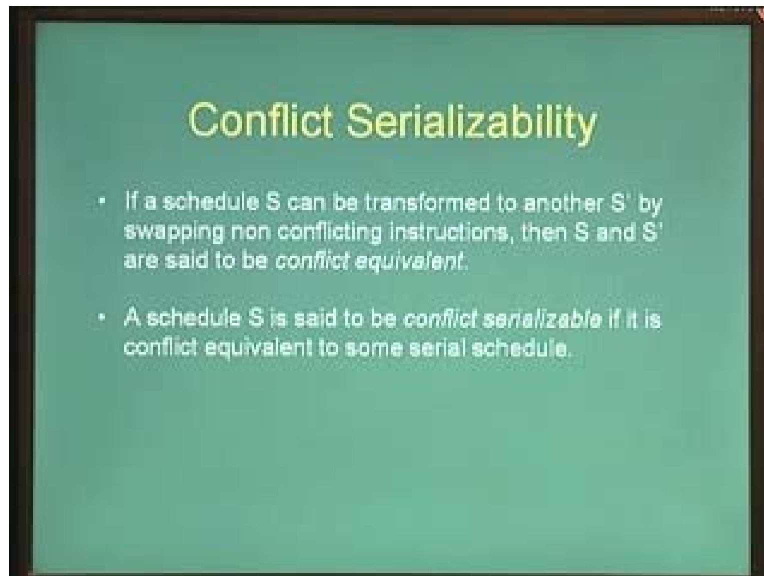


Let us look at the concept of serializability again where we talked about what is meant by serial schedule and how do we know whether a serial schedule is valid or not. In order to determine whether a serial schedule of transactional activities that are interleaved between one another, in order to know whether this is valid or not we have introduced the notion of conflict serializability. As, if you remember conflict serializability is a mechanism which defines the notion of conflicting database activities.

What are conflicting activities? Consider two activities I and J belonging to two different transactions T1 and T2. Now I and J can be executed in any order that is I before j or j before I doesn't matter if I and j refer to different data elements because they don't affect one another. And I and j could still be executed in any order, if they refer to the same data elements as long as both of them are just read operations. So both of them are just reading the given data elements, so it doesn't really matter whether I read the data data elements first or j reads the data elements first. On the other hand if either I or j or both contain a right operation on the same data element and both of them of course refer to the same data element then they are said to be conflicting. So we cannot swap the execution of I and j and expect that the swapping is an equivalent schedule to the earlier schedule.

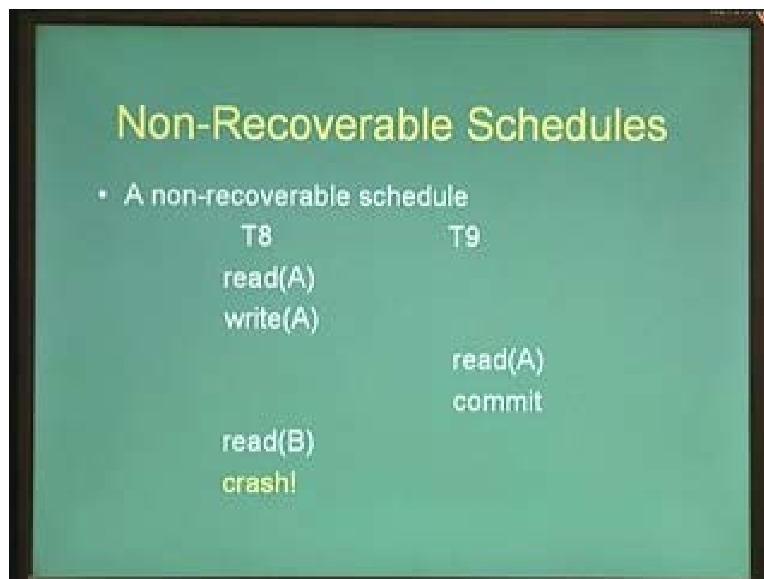
So if I have a schedule of operations, database update operations I can verify whether the schedule is safe or not by seeing whether it is conflict equivalent. That is can I keep rearranging the operations of this schedule so that I eventually end up in a serialize schedule that is all activities of one transaction are performed before the activities of the second transaction. So I end up in a serial schedule without encountering any conflicts during the way.

(Refer Slide Time: 16:25)



So if a schedule can be transformed in such a way or if it is conflict equivalent to a serialize schedule then it is said to be a conflict serialize schedule.

(Refer Slide Time: 19:07)



We also saw the notion of non recoverable schedules that is in what cases, we can never recover from a crash and so on. So the slide here shows an example of a non-recoverable schedule. There is a transaction T8 which is reading a data element A and writing something back on to A. That is it has performed some computation say as long as when we are concerned about recovery, we are not really concerned about what kind of

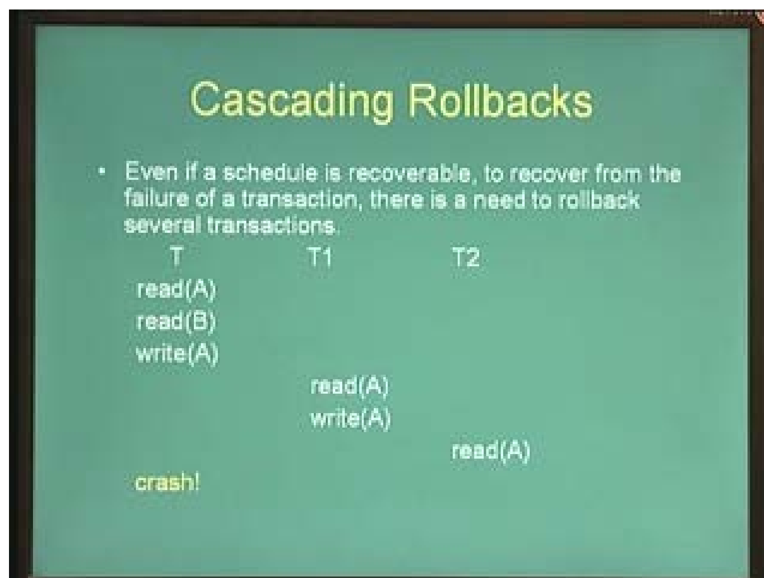
operation it makes. As long as there is some right operation we assume that some update has taken place, may be there was no update that is it has just read the data elements and return it back for whatever reason but at the level of recovery mechanism… (Refer Slide Time: 17:15) some change or there is some modification that has happened. So this transaction T8 has read a data element A and it has returned it back onto the data base.

Now after it has returned it back onto the data base, another transaction T9 read that element of A and of course did something and then committed. So this commit operation could probably involves some kind of physical operation like say displaying the data element may be it is the new stock prize or whatever, it just displayed the data element.

However this transaction T8 try to do something more and crashed. Now because transaction T8 has crashed, it has to be rolled back that is whatever operations that are performed by T8 has to be rolled back but we cannot roll it back because transaction T9 which has already read the changed data element has already committed. And whatever data that is returned is already out in the open and it’s been displayed. So such a schedule is a non-recoverable schedule. And how do we prevent non recoverable schedules from occurring?

Simple way of preventing non recoverable schedule is from this example is to note that transaction T9 cannot commit until transaction T8 has committed. That is if a transaction is reading a data element that is written by some other transaction then it cannot commit until the previous transaction has committed. So in that way transaction T9 cannot display the value of data element A and until and unless transaction T8 has successfully completed.

(Refer Slide Time: 20:36)



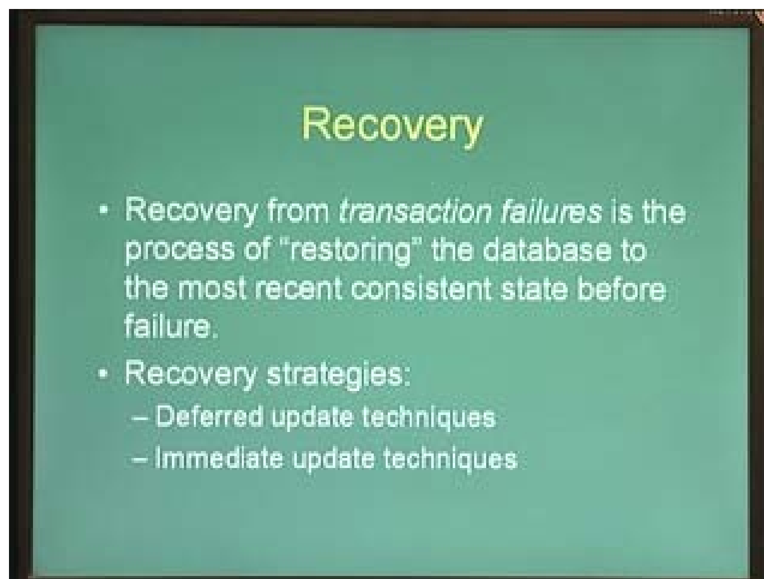
But even then that is even if we stipulate that transaction T9 cannot commit until transaction T8 has committed… (Refer Slide Time: 19:18) cascading roll backs. That is shown in this slide here. That is there are 3 transactions T, T1 and T2 and this transaction T has read data element A and modified it and returned it back into the database.

Now this modified data element now is read by T1 and then T1 in turn modified it again and wrote it back in to the database. And T2 then read this second modified database, second modified element that is the data that was modified by T1 and then probably try to display it or something. And of course, because we have ensured that none of them can commit until T can commit, they are just ready and waiting for performing whatever their commit operation tells them to do that is whether you display it or dispense money or whatever.

Now transaction T instead of committing crashes for whatever reason. Now because transaction T has crashed, transactions T1 and T2 even though they have completed successfully have no option but to roll back. So this is the problem of cascading roll back. so even if the schedule is recoverable, sometimes suppose transaction T is a long running transaction, it runs for several minutes or probably sometimes even several hours transaction T1 and T2 are short transactions and there are several such transactions are waiting on transaction T2, for transaction T to commit.

Now for whatever reason if the T transaction crashes whether it is a transaction failure or a system failure or a media crash or whatever, we have to roll back and all these transactions that are been waiting on transaction T.

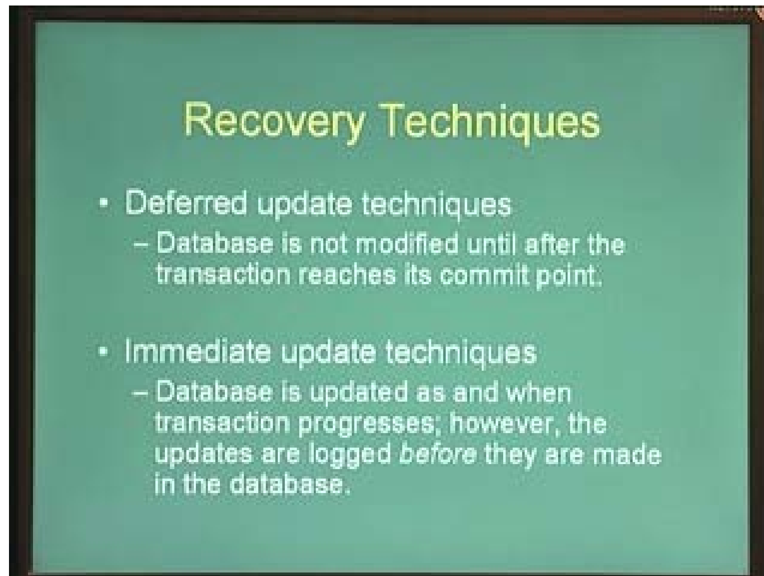
(Refer Slide Time: 21:17)



So let us look at how to tackle all these problems in a systematic fashion, so the concept of recovery. Recovery from transaction failure is a process of restoring the database to the… and where do we restore it? We restore it to the most consistent state that was there

before the failure. And there are two kinds of recovery strategies that we are going to be seeing today which are called the deferred update strategies and immediate update strategies. Deferred update essentially means that the database or updations to the data base are deferred until after sometime which will formalize later on. And immediate update techniques update the database as and when transactions are running. The database is physically changed as and when transactions are running.

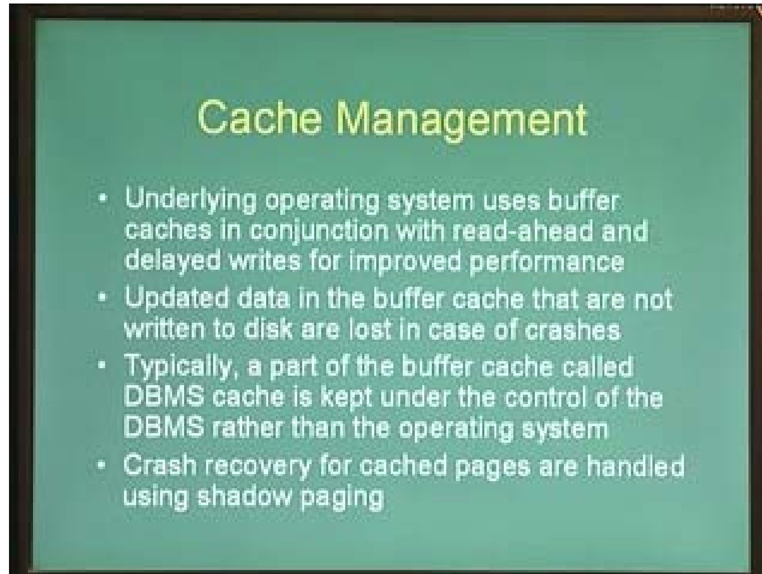
(Refer Slide Time: 22:20)



So this slide defines both of these techniques. Again the database is not modified until a transaction reaches it commit point in deferred update techniques and in immediate update techniques database is updated as and when transaction progresses. However transaction fails in immediate update techniques, you have to undo this operation that is you have to change this. Therefore they have to be logged whatever update were made to the database have to be logged before the updates are made. Obviously you can't log the updates after making the updates because what happens if the system crashes, once you have made an update and before writing the log.

On the other hand if you have written a log and the system crashes before making the update, it is still not a so much of a problem as we will see later.

(Refer Slide Time: 23:15)



Before we go on to recovery techniques, there are two things that we have to define and we have to be careful about how these impacts recovery techniques. The first issue is that of cache management. You might have studied in an operating systems course that most operating system use what is called as buffer caches. And what are buffer caches? Buffer caches are some buffered areas in memory that act as a cache for data that are present on disk. That is whenever a disk block or a disk sector is accessed or is sort by the operating system instead of just reading one disk block, usually operating systems perform what is called as read ahead that it reads a set of blocks into main memory.

And all writes that are performed on to disk sectors are initially performed just on the main memory and not onto the disk. And its only once in while these cache or this buffer cache is flushed onto disk. This is done in the interest of performance that is why for example in most operating systems, you need to perform some kind of disk sanity checks if the operating systems crashed midway because not all blocks that have been modified would have actually been written on to disk.

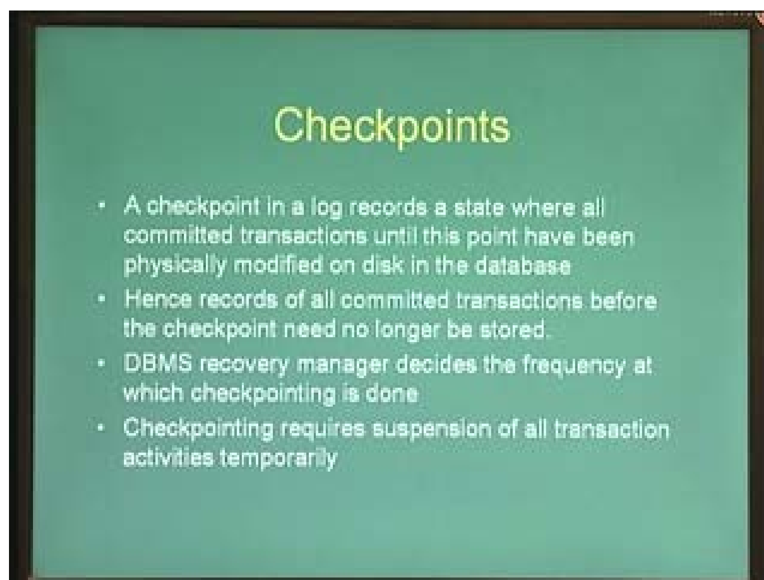
Now this buffer cache is an operating system construct that is it is in the control of the operating system. And application programs or user level programs that run in an operating systems usually don't have control for this buffer cache but for database recovery we need to have control over this cache because we can't assume that the operating systems has written something onto disk after we have said write because operating systems in turn has its own mechanisms that might defer writings on to disk and which may impact a recovery process.

Therefore typically in many database management systems, what is done is a part of the buffer cache that is maintained by the operating systems is given to the dbms. That is the dbms is given control of this buffer cache so that it can, that is the dbms can control … (Refer Slide Time: 25:48) into the buffer cache and so on.

And such kinds of cache pages which are given to the dbms are called dbms caches. And of course there is also the problem of what happens if the system crashes when the cache has being written on to disk. That is I have written something onto cache and now I am flushing this buffer cache but during this buffer cache flush, the system crashed and what do we do. The cache is partially written and the data could be inconsistent and so on.

For that a technique called shadow paging is used which we are going to study in the next session on database recovery technique. The shadow paging technique that is used for data base recovery can also be used for maintaining or recovering cache contents in the case of crashes.

(Refer Slide Time: 26:50)



The second issue that we are going to be concerned about is the concept of … (Refer Slide Time: 26:55) log. Now we have mentioned in passing that in order to aid the process of recovery from databases we maintain logs that is whatever updates are made to the database are all logged in some log file. Now this log file keeps on growing because every update that is made to the database, the information about this update is kept in this log files.

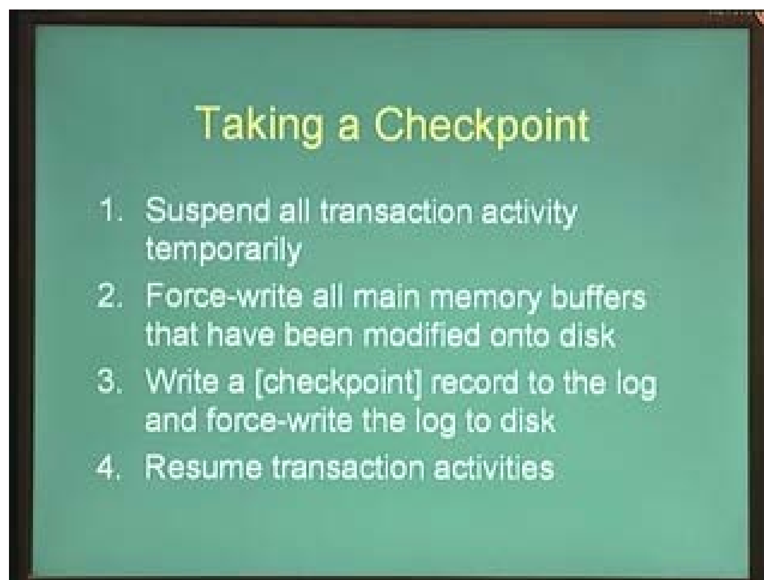
Now this log files keeps on growing and we don't know when a crash would occur and how much of history information we need and so on. So how do we prevent this log from growing forever and probably becoming bigger than the database itself. So the answer to this is the notion of a check point. A check point in a log records a state where all transactions that have been committed until this point in time have been physically modified on the disk in the database. That is the database has been updated and everything is fine for all the committed transactions until a check point.

So all committed transactions that have been, information about whom have been stored in the log until a check point can be thrown away. That is at a check point we can throw

away data about all the committed transactions that have happened before the check point. And at what frequencies do we have to check point the log? That is check pointing rather that is the process of introducing a check point in a log as you might have imagined is a separate process by itself that is we have to decide at what intervals at or at what frequencies are we going to introduce check points into the log and what should be done when a check point is being introduced.

And check pointing actually involves suspension of all activities of the database, all transaction activities of the database temporarily until we know for sure that all the that this check pointing criteria is made that is all the committed transactions have been successfully updated onto the disk.

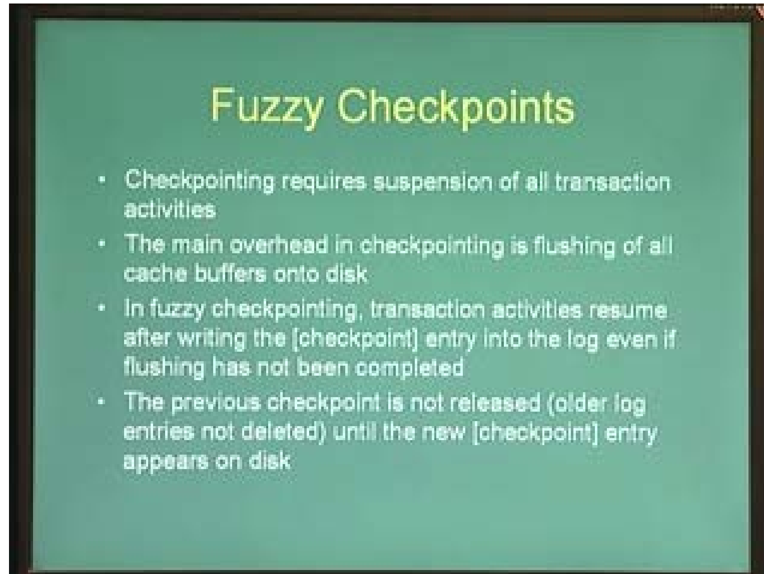
(Refer Slide Time: 29:10)



The algorithm for check pointing is quite simple but quite costly in terms of operations that is in order to take a check point, we first suspend all transactional activities temporarily because we don't want more data to be written when we are handling this check pointing. And then we force write that is we flush all main memory buffers that have been modified to disk that is whatever has been, whatever data that had to be updated onto the database we force write all of these committed … (Refer Slide Time: 29:47) and then we write a check point note in the log file and and also of course force write this log on to disk.

The fact that we have written, the fact that we have encountered a check point should also be recorded persistently onto disk because once we have thrown away information about other transactions, we can't lose the fact that we have performed a check pointing operation and then we resume transaction activities.

(Refer Slide Time: 30:20)



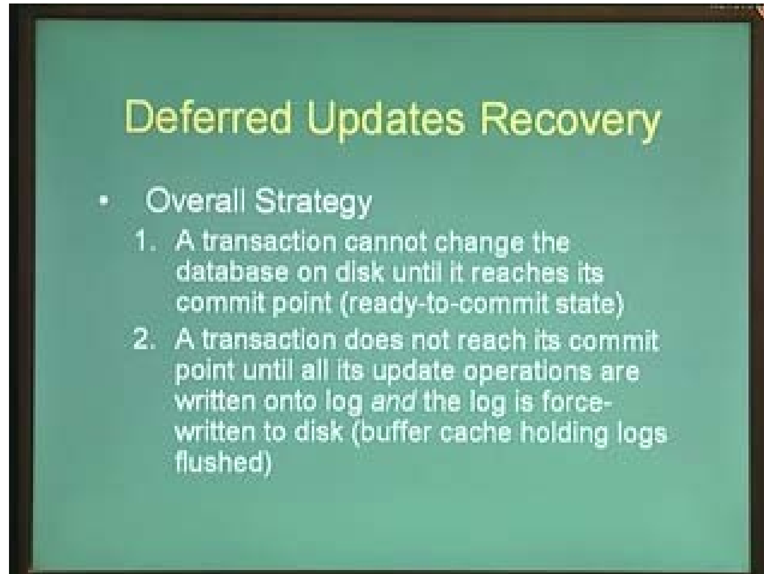
There is also a notion of fussy check points where that are more, slightly more efficient than the usual check pointing techniques. Note that check pointing requires suspension of all transaction activities and if this is done too frequently then it impacts database performance itself.

Now let us see where is the biggest overhead during a check point and seek can we do something about making this check pointing faster. The main overhead in check pointing and I am sure you would have imagined that is the flushing of all the buffer cache onto the disk. That is each buffer cache contains set of disk blocks and so all of these disk blocks have to be physically flushed on to disk and this is what is going to take the most time.

Now in fussy check pointing what happens is that transaction activities resume after writing the check point entry into the log, even though flushing has not been completed. That is even the check pointing, check point entry could be in the cache and all the flushing actives activities are still going on but transaction activities, the transaction activities resume. However the previous checkpoint is not released that is the older log entries are not deleted until after the new check point entry has been flushed onto the disk.

So it's a background operation where until we are sure that the new check point operation has been written on to disk, this can be written onto disk only after all the buffer cache buffer has been flushed on to disk. So until we are sure this has been done, the previous set of log entries are not deleted.

(Refer Slide Time: 32:11)

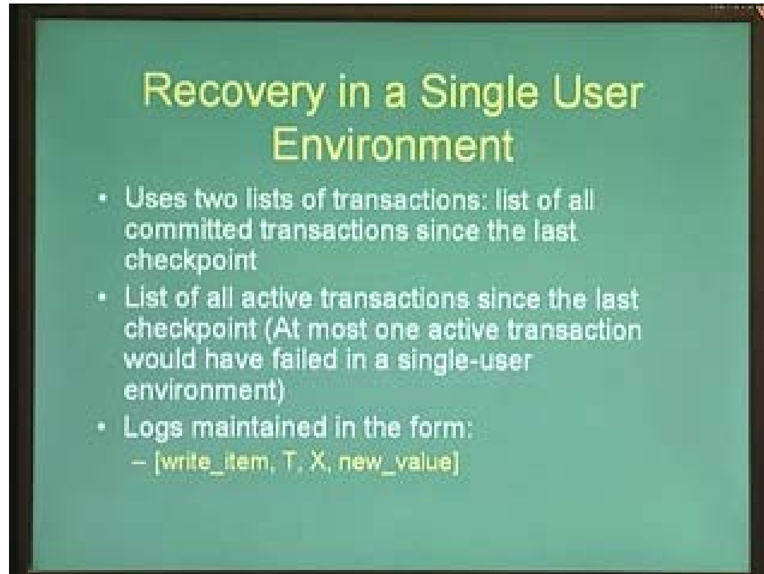


So let us look at the first kind of recovery technique which we called as the deferred updates recovery. So what is the notion of a deferred update recovery? As the name suggests deferred updates means that the updates to the database are deferred until transaction commits that is until transaction has reached a ready to commit state.

The overall strategy for a deferred updates recovery is simply this thing. That is a transaction cannot change the database. So even if a transaction has run half or 90% or 95% or whatever, it has not made any changes on to the data base as far as until it has committed. So until it reaches the commit point, the database is not updated. And a transaction does not reach its commit point until all its update operations are logged and the log is force written on to disk.

That is the transaction does not say am not ready to commit until all its operations that have been done have been logged and this log is available on the disk. Therefore even if the transaction crashes now and even if the database is not updated, we have the log entries which says that these are the changes that were made and the transaction is now ready to commit and it can commit.

(Refer Slide Time: 33: 42)

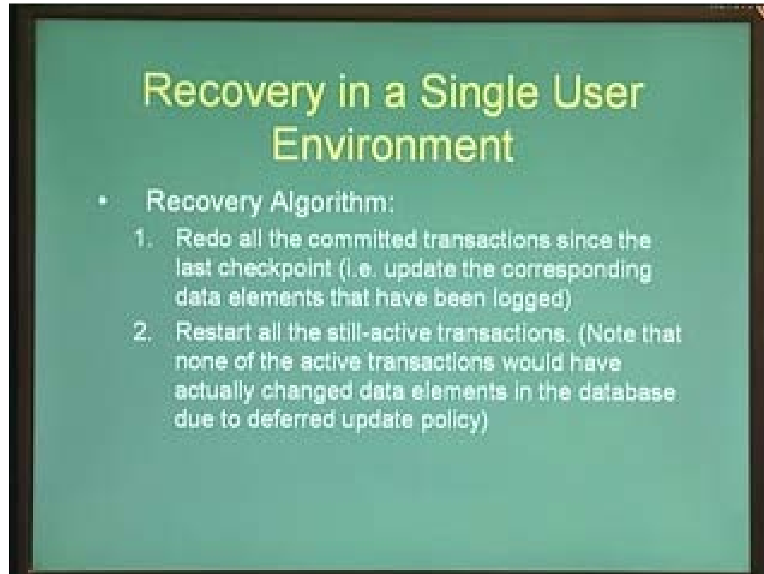


So using this let us see how we can perform recovery. We first look at recovery in a single user environment. That is it is a sequential database engine where transaction… (Refer Slide Time: 33:52) transaction is performed one after the other. This is a simplistic case but it helps in understanding how the deferred update technique works. So deferred update techniques uses two lists of transactions. That is one is the list of all committed transactions, when does it use the two lists of transactions. That is after, it is after the disk that is after the system has been brought up following a crash and the system is asked to recover to a consistent state.

Now once the system is asked to recover to a consistent state, the recovery process starts by using two lists of transactions. One is the list of all committed transactions since the last check point and the list of all active … (Refer Slide Time: 34:42). Since it's a single user operation there would be utmost one such transaction which would have failed in an active state. And a logs that are maintained for these transactions are maintained in the following form which is shown here. That is the first element here says that this is a write item that is something has been written on to disk.

Note that as far as recovery is concerned, we are interested only in these write items, logs can be used for a variety of purposes something like to understand the behaviour of transaction, to profile the performance of the data base and so on and so forth. But as far as recovery is concerned, we are just interested in what changes have been made to the database. Therefore we are interested in only these write items. So it says this is a write item belonging to transaction T involving data element x and this is the new value that was written on to the x or that has to be written on to the database for element x.

(Refer Slide Time: 35:49)



Now once these two lists are maintained that is the list of all committed transactions since the previous check points and the list of all active transactions. we first start by re doing all the committed transactions, that is we take the set of all committed transaction since the previous check points and then go about looking at the logs and see what values they had written on to what data elements and we start writing those values once again.

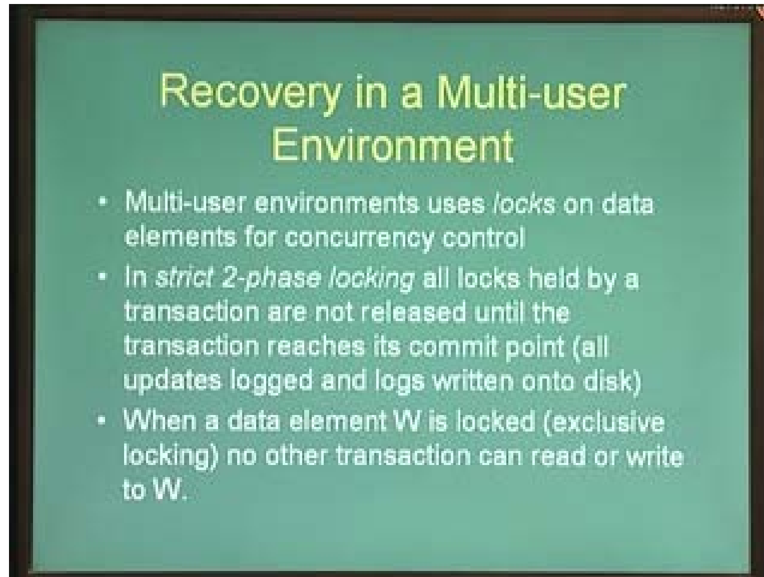
We don't care what is the semantics of these values or whether these values were read or anything because these transactions were already committed, the cash has been dispensed already. so we don't have to do any physical operations, we don't have to even tell the application program that we are doing these things because these were already committed and we know what are the values that has to go into the database, we just write those values. we just start from the previous check points and start, data element x has to have a value of 10, data elements y has to have a value of abc or whatever and we just start writing those values back in to the database.

And then for all the active transactions which crashed midway … (Refer Slide Time: 37:12). During execution we just have to restart all those active transactions because none of the active transactions are physically modified the database that is what the deferred update technique all about. That is transactions don't modify the database until they are ready to commit and they won't be ready to commit until they have made all the log entries on to disk.

After they have made all the log entries on to disk and said that they have been committed then they have been treated as a committed transaction and the first… (Refer Slide Time: 37:48) and all the redo operations from the transaction is performed. If for any case the transaction that is the ready to commit tag does not go in to the log or whatever has been written by the transaction is not flushed on to the log and the system crashes then it is treated as an active transaction and it is just started once more. And it

runs once again and makes changes on to the logs and if everything goes well that is the log is flushed on to disk then it is ready to commit and the data base is updated.

(Refer Slide Time: 38:28)



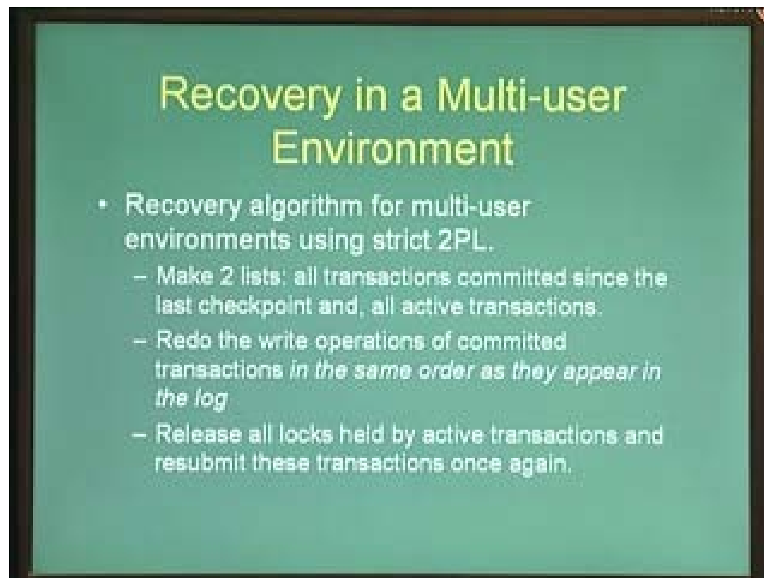
What about updates if the database is actually a multi user environment and there are several concurrent transactions that are running at the same time. In a single user environment, we don't have to worry about concurrency control that is how transactions are serialized that is they run in a serialized fashion by default. But in a multi user environment there is the problem of serialization of transactions. Now serialization of transactions is not the problem of the recovery part of the database, it's usually handled by the concurrency manager that is whatever concurrency control techniques are used for managing concurrency.

Now we assume that for deferred update techniques, we require that concurrency control uses what is called as a strict two phase locking. You might have heard of the notion of locking in several contexts in operating systems and probably even in systems design and so on where locking essentially means that if a transaction is performing some some updates on a database or some data elements, it obtains a lock. That is it locks the data elements so that nobody else can read or write to the data element or can access the data element. Of course there are two kinds of locks that is read locks and write locks so read locks can be shared but write locks or exclusive locks cannot be shared that is once a transaction has obtained a lock on a data element that it is going to modify, no other transaction can even read the element until the lock is released by the transaction.

So in strict two phase locking all the locks that are held by a database or by a transaction are not released until the transaction reaches its commit point. Now what does it mean that it reaches its commit point? That is all the updates it has made are logged that is are written on to logs and the log is flushed on to disk. now the transaction is sure that somewhere whatever updates it has made is persistently stored that is the updates it has

made is safe somewhere, its not just lost once if the system crashes only then it will release its locks. So only after it releases its locks can other transactions read the data element, read the corresponding data element.

(Refer Slide Time: 41:08)

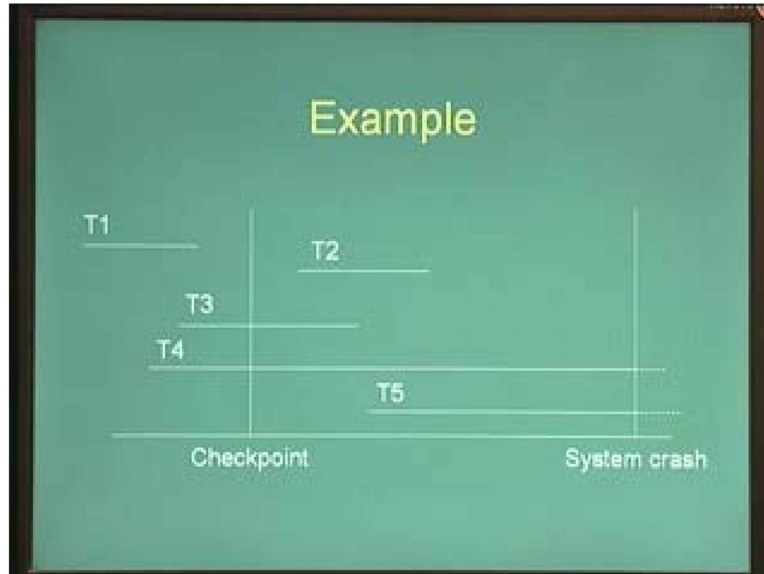


So assuming that we have strict two pl that is strict two phase locking, the recovery process can follow the technique shown here. Firstly make two lists of transaction, once after a database has crashed and it has restarted and the recovery algorithm is begun and it is starting to recover, make two lists of transactions. That is first is the set of all transactions that have committed since the previous check point and the list of all active transactions.

Now for all the set of transactions that have committed, we have to redo the operations. This is the same thing we have done in the single user environment. However here we have to ensure, we have to explicitly state that redo of the operations are performed in the same order as they appear in the log. We cannot try to optimize this redo operations by introducing some concurrency there because they may violate some kind of serializability conditions if they are performed in some other order … (Refer Slide Time: 42:24) between two or more updates such that they cannot be swapped in their ordering.

And once the redo operations are performed that is once all the committed transactions have been persistently written on to disks, we then restart all the active transactions and before that we release all the locks that have been held by this active transactions and again the ground is free so that whoever wants the locks can now hold those locks. So we release all those locks and resubmit all the active transactions once again.

(Refer Slide Time: 43:01)

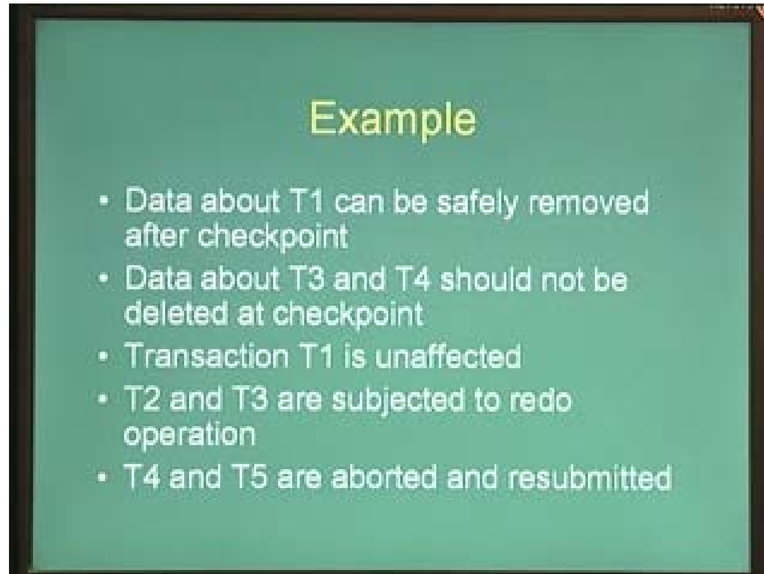


Let us take an example of different transactions and see what happens in a multi user environment. So this slide show a set of transactions and two different events, one is a check pointing event that happens here and one is the system crash that happens here. Now there are several transactions in a database here. Transaction T1 has already committed before the check pointing event happened. Therefore after the check pointing data about T1 are thrown out, we don't even need data about T1 anymore. However T3 and T4 have started before the check pointing operation but they have not completed, therefore we cannot throw away these data about T3 and T4 even if check pointing is performed. And transaction T2 has started only after the previous check point but has committed its operation before the system crash.

While T4 and T5 have not yet committed, T4 is really a long transaction that is taking place and they have not committed when the system has crashed. So what happens in the update operation here that is during the deferred update operation? T1 is not concerned at all, it doesn't figure in to the picture at all because there is no data about T1 so at the time of the updates that is happening here … (Refer Slide Time: 44:31) committed but the data is still there because it's occurring after the check point. And because it has committed transaction T2 will undergo redo operation that is all its updates are logged into log file and using the log file, the database is updated. Transaction T3 has also committed before the system crash, it has started before the check point therefore its data will not be thrown away and because it has committed before the system crash this is also re done. That is redo operation will be performed on transaction T3.

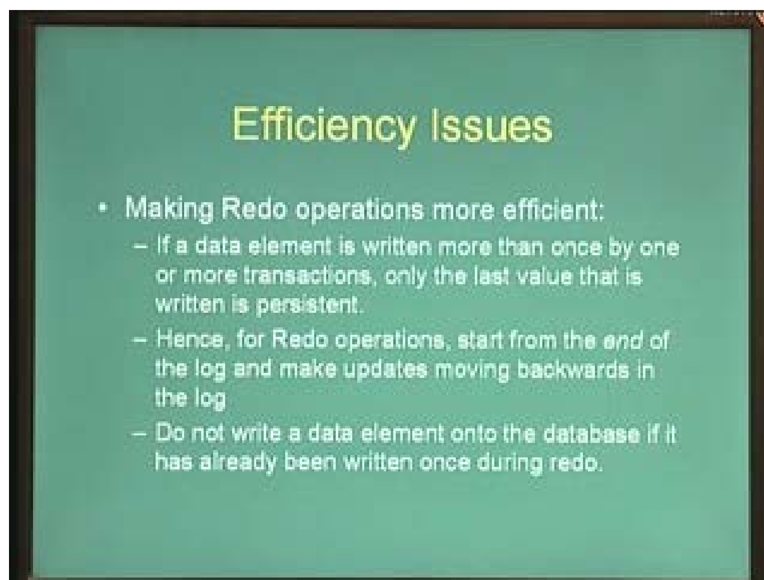
And transaction T4 is still active at the time of system crash as this transaction T5. Therefore both T4 and T5 have to be resubmitted to the dbms that is after releasing all their locks that they have held, they have to resubmitted back to the dbms.

(Refer Slide Time: 45:27)



So that's what this slide says that is data about T1 can be safely removed after check point, it doesn't even figure during system crash. That is it doesn't even figure during updates recovery rather. And data about T3 and T4 should not be deleted at check point because they are not committed and transaction T1 is unaffected during the recovery process. T2 and T3 are subjected to redo operations and T4 and T5 were aborted and resubmitted back into the dbms.

(Refer Slide Time: 46:00)

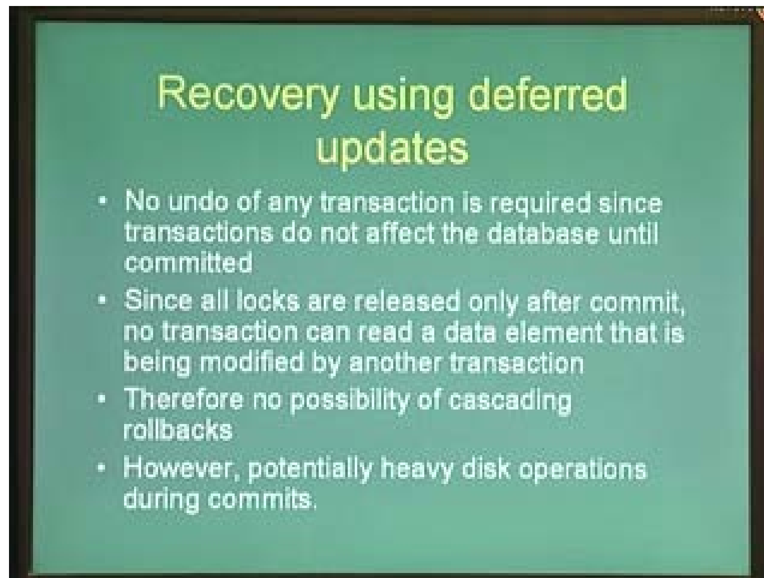


There is some efficiency issues about redo operations. small thing we can notice, if a data element x has been written (Refer Slide Time: 46:15) it is enough if we just write the last

update on to the database because anyways if we write a previous update, it is going to be over written by the next updates.

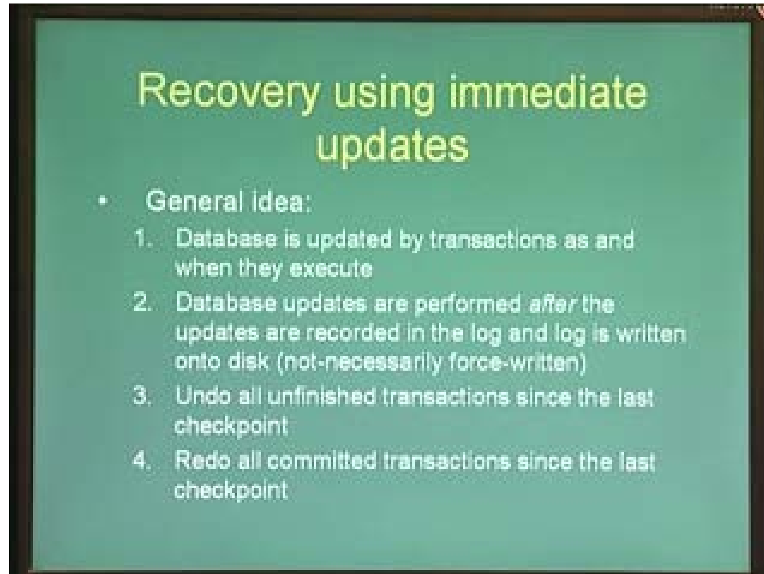
Hence for redo operations we can start from the end of the log and start making updates moving backwards in the log. And we should not write a data element on to the data base if it has already been written once during redo because we have already written the latest value during redo.

(Refer Slide Time: 46:45)



What are some of the properties of deferred updates? There is no undo that is required. If you have noticed we have only talked about redo, there is no undo operations that are required. Why? Because the database is not modified at all, it's only the transaction logs that are modified. And since all locks are released in strict two pl, all locks are released only after commit. A transaction can read a data element that is being modified that is no transaction can read a data element that is being modified by another transaction. Therefore there is no possibility of cascading roll backs because one transaction has already read a data element that is been modified by some other transaction and waiting for it to commit. So there is no such possibility, so there is no possibility of cascading roll backs. However potentially there is a large amount of disk operations during commit because enormous amounts of updates in especially large transactions that have been written on to logs have to be written back on to the disk.

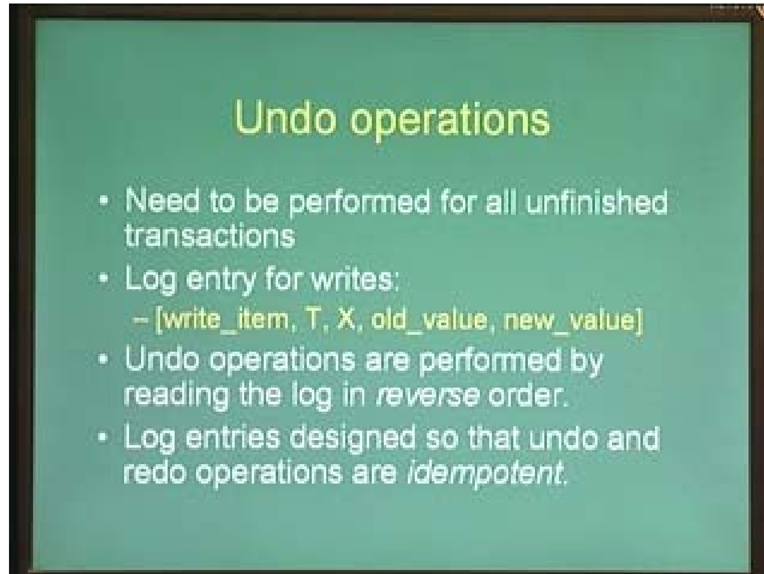
(Refer Slide Time: 47:53)



The next technique that we are going to look at is what is called the immediate update techniques. In immediate update techniques, the database is updated as and when transactions execute. However for the sake of recovery, database updates are performed after the updates are recorded in the log and the log is written on to disk. That is only after we know that is there is still some kind of deferred updates happening here, that is updates are deferred only after we know that the log is written on to disk and before which we modify the database.

As and when we know that a particular log entry has been written on to disk, all those corresponding entries can be modified. And however in the phase of a crash, we have to undo all un finished transactions since the last check point. We need an undo operation here which is not required in the deferred updates. And we still need the redo operations for redoing all the activities of the committed transactions since the last check point.

(Refer Slide Time: 49:03)

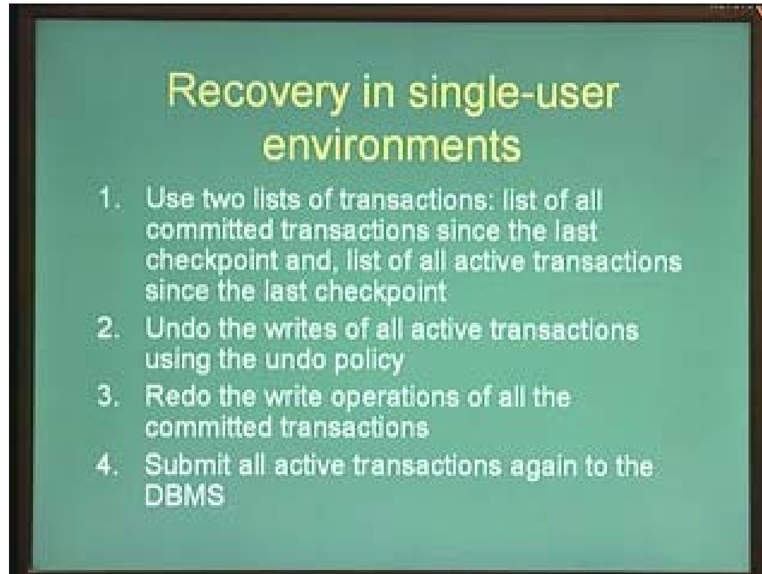


So let us first look at the undo operations that is what should be done for all the un finished transactions. Now in order to perform undo, we need an extra element in the transaction logs. This is shown here that is a transaction log contains this write element, write item element and and transaction t that is this element belongs to transaction t involving data element x and it says that the old value of x was this and the new value was this. Therefore when we are undoing it, we have to replace x by its old value and because we dint have an undo, we dint have to store old value in deferred updates.

And of course undo operations have to be performed in the reverse order obviously because the oldest values have to remain on the database. And these log entries, the way these log entries are created is such that undo and redo operations are what are called as idempotent operations. What is an idempotent operation? (Refer Slide Time: 50:11) idempotent operation is something where it does not matter how many times you perform the operations.

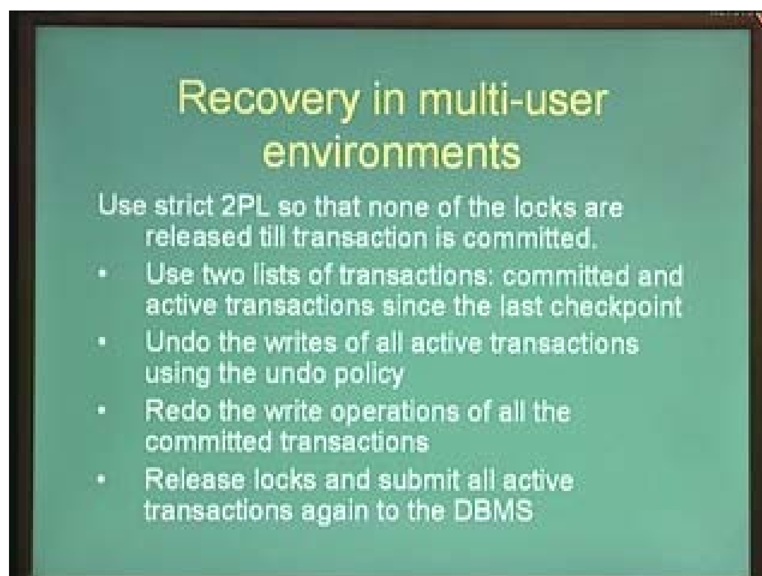
For example some problems during undo and undo couldn’t finish, we can just restart this undo process from the beginning once again and then run it again. And it doesn’t matter because it’s just re writing the old values, its not performing any computation. It’s not saying that value of x was changed by 5% so reduce it by 5% or something like that. So it’s not performing any computation, it’s just re writing back on to disk.

(Refer Slide Time: 50:44)



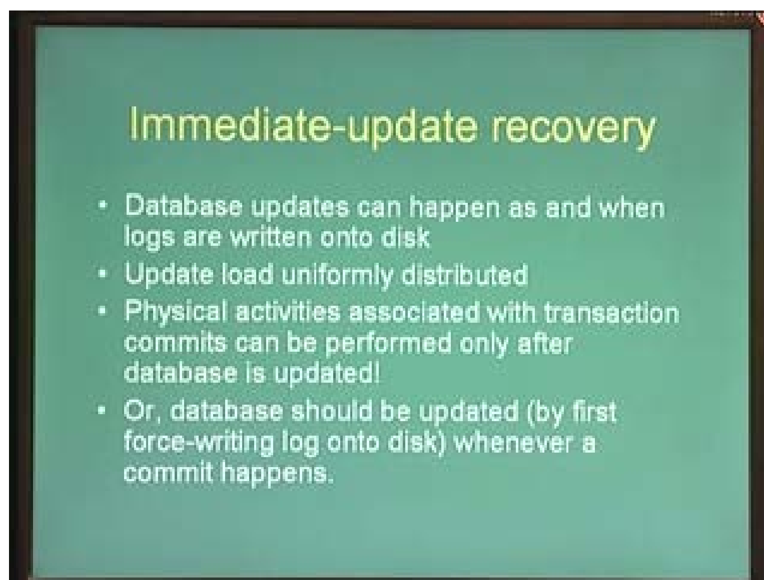
So how do you perform recovery in single user environment using immediate updates? Again like deferred updates we use two lists of transactions, list of all committed transaction since the last check point and list of all active transaction since the last check point. And first we start by undoing the activities of the set of all active transaction using the undo policy that we just saw in the previous slide. And then we perform the redo operation of all the transaction that have been committed since the last check point and then we submit all the active transaction back to the dbms so that can execute once again.

(Refer Slide Time: 51:32)



And how do we perform recovery in multi user environments using immediate update techniques. It's the same, it's quite similar to that of the deferred update techniques where we use strict two phase locking so that none of the locks are released until the transaction is committed. Therefore there is no possibility of a cascading roll backs. And as before use two list of transaction that is the list of all committed transactions and the list of all active transactions since the last check point. Then undo the writes of all active transactions using the undo policy and redo the write operations of all the committed transactions using the redo policy. And then just release all the locks that have been held by the active transactions and give the transactions back to the dbms that is resubmit the transactions.

(Refer Slide Time: 52:28)

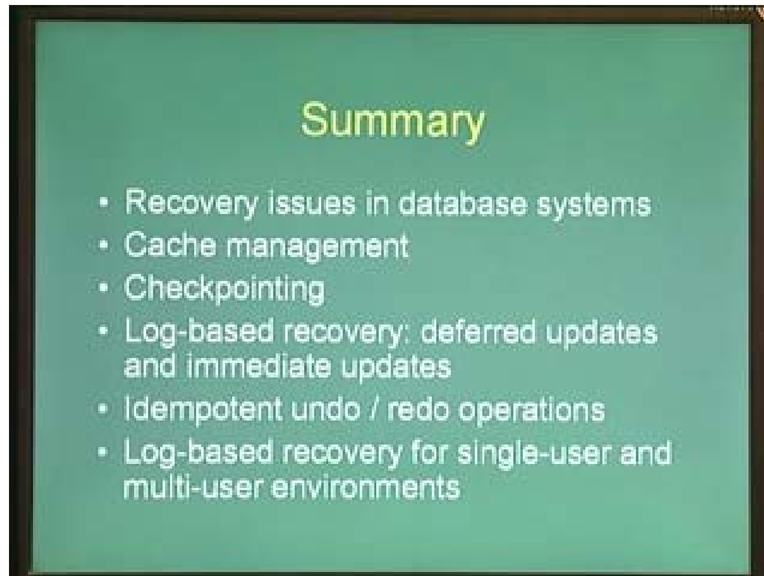


So what are the properties of immediate update recovery? As you can see database updates can happen as and when logs are written on to disk. That is the operating rather the dbms should only keep track of when the buffer cache logs are written on to disks. So as and when the buffer cache logs are written on to disks, the appropriate database update can also start happening. So because of this database updates, the load on database updates is uniformly distributed they are not burstive as in deferred update transaction, deferred update techniques where all updates happen at the commit point.

But note that any physical activities that is being performed by the application program like say dispensing money or launching a missile or whatever, so any physical activities that is to be performed by the upper application program can be performed only after the database is updated that is only after we know that… (Refer Slide Time: 53:37) and the database is updated. Why? This is because even if transaction is committed, the fact that it has been committed might still be in the buffer cache, it may not be written on to disk and the system could crash. and once the system comes back again, it is treated as an un committed transaction and it is run once again …(Refer Slide Time: 54:05) even if in memory the dbms knows that it has been committed, it cannot or it should not tell the

application program saying everything is okay, everything is still not okay. It will be okay only when the the disk is updated that is once they are flushed on to disk, so commits can be performed only either the database or the logs are updated. That is force writing logs on to disk whenever a commit happens.

(Refer Slide Time: 54:34)



So that brings us to the end of this session on transaction recovery using log based recovery techniques. Here we saw several different issues; we started with several different issues concerning recovery in database systems. We looked at the different kinds of failures that can happen and what it means to recover from a system crash or some kind of a failure.

And there are two issues that affect recovery processes. One is cache management that is for the sake of efficiency disk blocks are usually cached into ram in the buffer cache. And because we are talking about recovery, this can impact recovery process because we are not really sure whether something that has been written on to disk has actually been written on to disk.

So some part of the buffer cache is usually controlled by the dbms and which is called the dbms cache. And then we looked at the concept of check pointing which allows us to throw away, safely throw away historical information that is stored in logs. And then we looked at two kinds of log based recovery techniques deferred updates and immediate updates both of which use what are called as idempotent, undo and redo operations. There is no undo operation in deferred updates but there are undo and redo operations in immediate updates. And then we also categorized this log based recovery into two different kinds, single user environments and multi user environments. And in multi user environments we have a requirement that we have to use strict two phase locking in order to prevent cascading roll backs in case of a crash recovery. So that brings us to the end of this session.