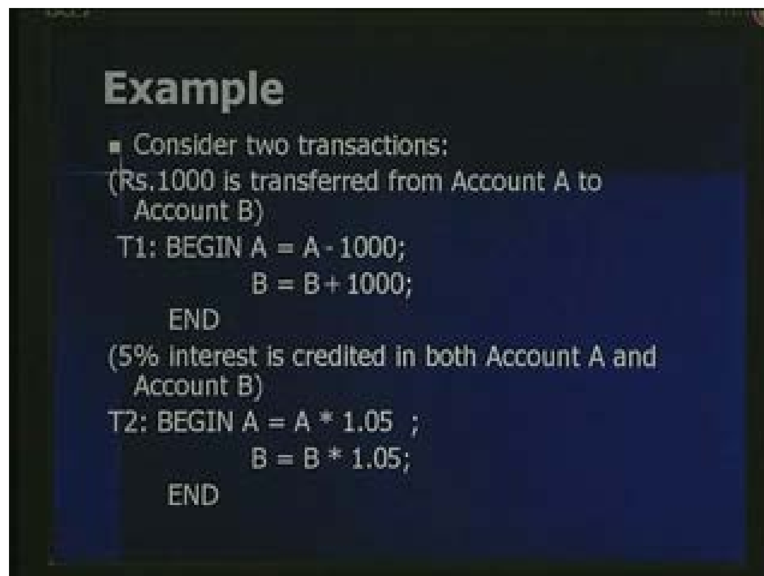
Database Management System
Prof. D. Janakiram
Department of Computer Science & Engineering
Indian Institute of Technology, Madras
Lecture No. # 18

Transaction Processing and Database Manager

In the previous lecture we have looked at the basic properties of transactions namely the acid properties atomicity, concurrency, isolation and durability. In today's lecture we are going to see how these properties will be realized by the transaction processing system within the database manager. We will take a few simple examples and through this examples we will illustrate how the transaction processing system will ensure the acid properties of the transaction.

[Refer Slide Time: 02:04]



Here is a very simple example shown in the slides here. There are two transactions here which are shown T_1 and T_2. T_1 is a account transfer transaction, transfer of money from one account to other account. Now 1000 rupees had been transferred from account A to account B by transaction T_1, transaction two is an interest payment transaction. So it is actually crediting into each account a 5 % interest into each of the accounts.

Now what is shown here is this T_1 and T_2 operating simultaneously on the banking system. Now what we will do is we will try to understand these two transactions in terms of the various operations performed by these transactions. What I am going to do here is I will write T_1 as performing several transactions or several operations. Now the first operation that is performed by T_1 is to actually take the account A and read the value of the balance that is there in this account. So it is basically a read operation of the account. The second operation is essentially to add 1000 rupees into this account and the third

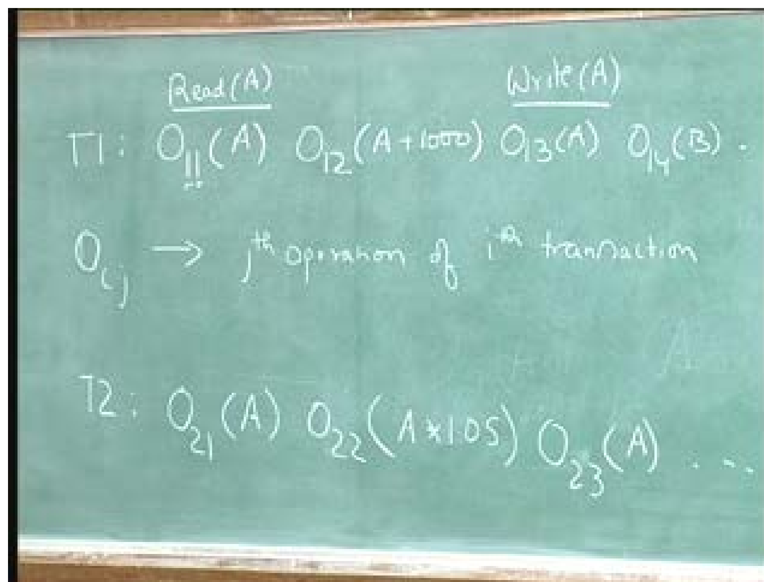
operation is going to be writing the value back. The same thing is going to be done for the account B. So I will actually show that as the fourth operation but the subsequent operations will not be shown here.

They are self-explanatory one can understand after that, the other operations. Now the suffix here shows that this is the first operation and this gives the transaction id. One is the transaction id and the one here indicates that it is the first operation. So operation 1 1 indicates that this is the first instruction of transaction. This indicates that it is the second instruction of transaction 1 like that it is shown here.

Now if you basically take a operation $O_{i\,j}$, it indicates that this is j th operation of i the transaction. This is the notation that we will be using. Now as you can see here this is basically a read operation on A and this is basically a write operation on A. So we have between the processing we have the reads and the writes happening on the data items. Now we can also understand the transaction T_2 also has shown in the slide as trying to do the following operations.

O_{21} is going to be a read of A and then the O_{22} operation is going to be an update on the value of the data item and then O_{22} O_{23} is going to be an item again A the rest of the operations as shown in the earlier case for the B, operation B.

[Refer Slide Time: 06:17]



Now what we are going to show you is what happens when these transactions are executed simultaneously on the database. Now a list of actions form a set of transactions as seen by the dbms. As you can see here O_{11} O_{12} O_{13} O_{14} are set of transaction, set of operations constituting T_1. Similarly we have O_{21} O_{22} O_{23} constituting the set of instructions constituting transaction T_2.

[Refer Slide Time: 06.52]

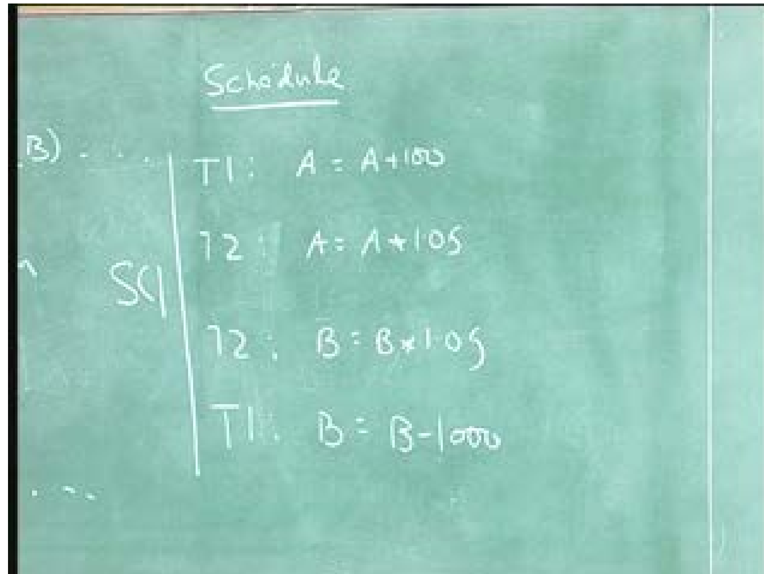
- A list of actions from a set of transactions as seen by the DBMS → Schedule
- Two Possible interleavings(Schedules)

Schedule 1	Schedule 2
T1: A = A + 1000	T1: A = A +1000
T2: A = A * 1.05	T2: A = A * 1.05
T2: B = B* 1.05	T1: B = B - 1000
T1: B = B -1000	T2: B = B * 1.05

Now it is possible for these operations to get interleaved in the sense that it is possible for these transactions, operations of the transactions to execute in an interleaved fashion. Now when this gets executed in an interleaved fashion, we basically call that as schedule. A schedule is nothing but a series of operations as executed by the database management system.

Now you can see here it is possible for these two transactions to execute concurrently, a set of interleavings that were shown is the operation T_1 are constituting A equals to A plus hundred is executed here. Then T_2 is executed which is equivalent to saying A equals to this operation is executed here, followed by the T_2 of B then T_1 again B equals to B minus. This is one schedule which is called a possible schedule, we can call this is as SC_1 as one possible schedule. As you can see here these two constitute transaction T_1 these two operations, these two operations constitute T_2.

[Refer Slide Time: 08.29]



Now there is another possible schedule also shown in the slide there in which case T_1 and T_2 are executed as shown here. But the other two operations are interchanged. This is executed before the other operation. Now when we actually have the schedules, one of the important criterion for this schedules to be valid is to see that these schedules produce proper consistent results at the end of the execution.

[Refer Slide Time: 09.07]

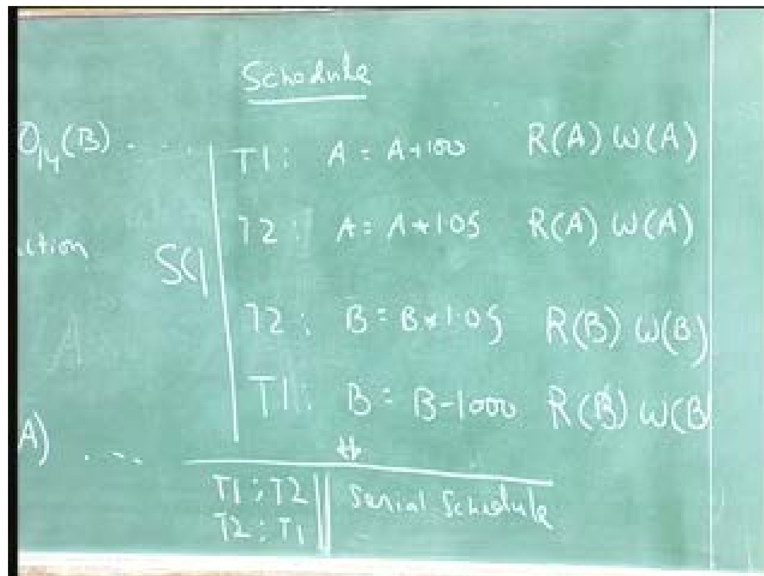
Now this schedule one can be seen as shown in the slide as read and the writes on the various data items. T_1 is basically is reading A and then writing A. It is reading the old value of the bank account balance of account A and writing the new value for the balance

here. T_2 is also doing a read of A and write of A because you are computing the new value of A by calculating the interest that is payable for this account. Similar way we can also write for write B and then read B and write B and read B and then again write B. What we are going to show you through this example is what happens when these reads and writes are interleaved from the consistency point of view.

Now we can see here the notion of what is correct from the execution point of view is shown here. Suddenly we don't want transactions to execute one after the other because then the throughput of the system will come down drastically. You want as many operations as possible should be executed in a concurrent fashion to actually increase the throughput of the system. Now when concurrently executed transactions at the end of it whether they can be translated into what is called a serial execution.

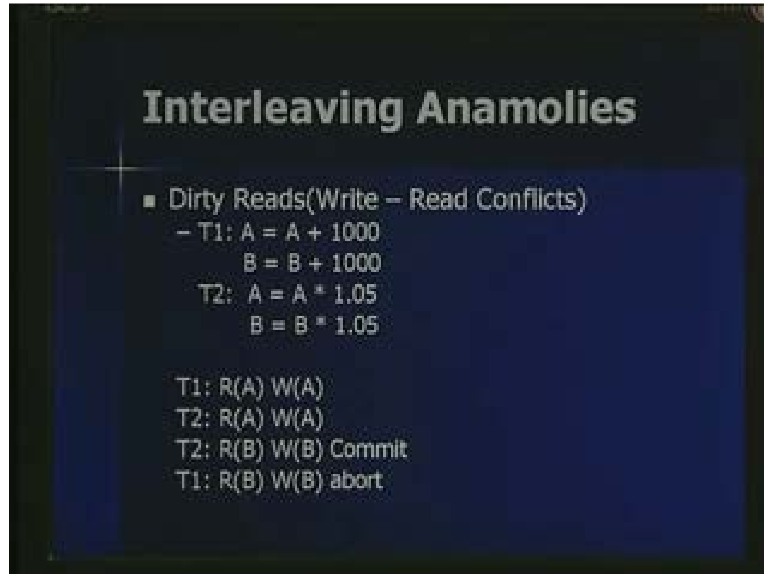
Now in this particular case, what you would like to say is T_1 executed completely after T_2 this is one possibility or T_2 executed after T_1. As long as this is possible for you say, we call this kind of schedule as a serial schedule. And that is what is actually shown in the diagram here. A schedule is which is equivalent to one of the serial schedules is equivalent to saying that either T_1 executed before T_2 or T_2 executed before T_1 that is one of these should be possible.

[Refer Slide Time: 11.28]



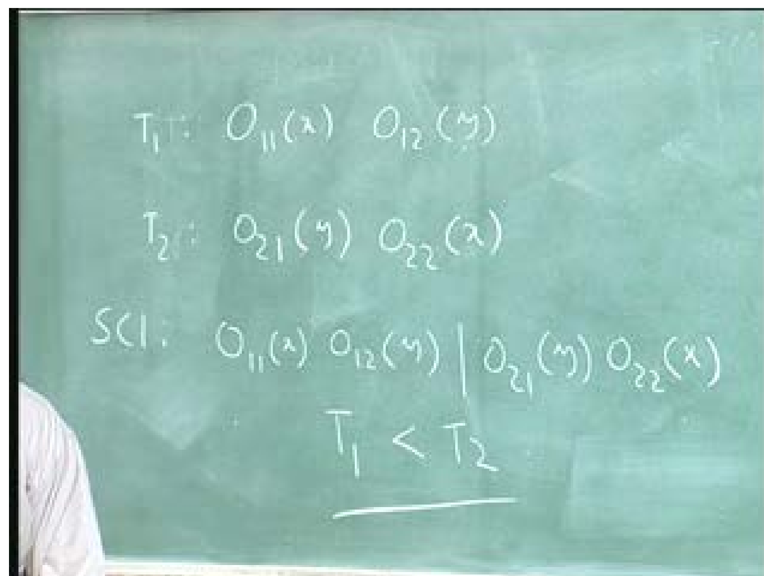
Now to see what really happens when the execution is not serial that means when finally you are not able to detect saying that the execution of the transactions is not as per the serial schedule.

[Refer Slide Time: 11.51]



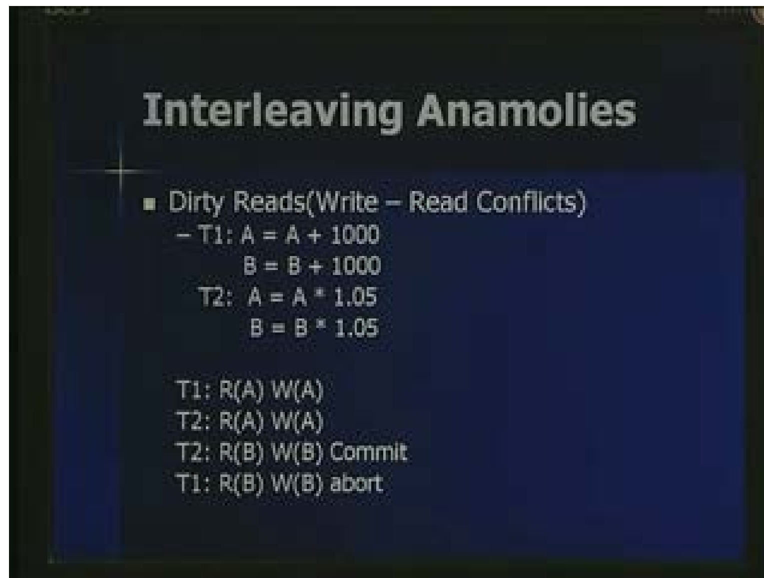
I will take a very simple example and show how exactly will deduce the serial schedules. Let us take a case of a transaction T_1 with operation x and then 1 2 which is actually a write of y. Now we can take another transaction T_2 where basically it is a read of y and then a write of x. Now no matter how these operations are executed. As long as it possible for you to say that all the operations at T_1 have been executed before T_2 which is equivalent to saying that if there is a schedule that says $O_{11}(x)$ and O_{12} (y) has actually finished before O_{21} (y) and O_{22} (x) are executed by the database manager. This is equivalent to saying that T_1 finished before T_2. This is what actually we mean by a serial execution of the transactions.

[Refer Slide Time: 13.26]



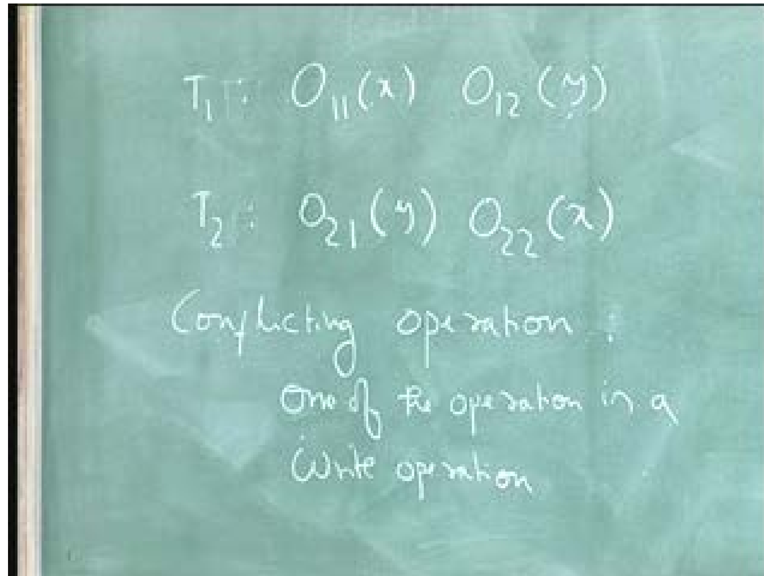
That is T_1 finished execution before transaction T_2 started executing. Now it is also possible for you to also have the reverse order where all the operations of T_2 have been executed before T_1. This is a very simple and straight forward case where we can easily save on all the operations of T_1 and T_2 are executed in this particular fashion very easy to see that T_1 has actually finished all the operations before T_2 stated executing. The only case where you will have problems is when some operations of T_1 have been executed in such a way that they are interleaved with the execution of the operations of T_2.

[Refer Slide Time: 14.12]



Then the problem of deciding whether the schedule is equivalent to a serial schedule that all the operations of one transaction finish before the other becomes a important requirement. And that is what actually we are going to look at how that can be done. In this particular case what we are going to say is all that will be required for us as a criterion whereas schedule is produced is two operations are said to conflict, we say the notion of a conflicting operation is when one of them is a write operation. One of the operations is a write operation.

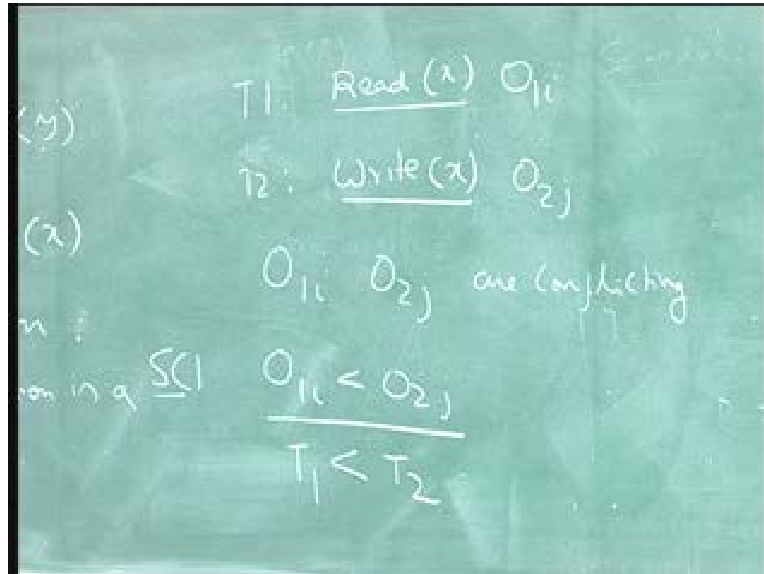
[Refer Slide Time: 15.15]



Now to give you the little more simplistic view, let us say T_1 is trying to read a data item x and T_2 is actually trying to write the data item x. These two operations O_1 of some i, O_2 of some j are said to conflicting because they are operating on the same data item and they are conflicting with each other. Here as you can see here, T_1 is reading the data item x, T_2 is writing the same data item x. Since both transactions are reading the same data item and one of the operations is a write operation, we say that these two operations are called conflicting.

Now whenever we have conflicting operations like this, the first inference is here O_1 i and O_2 j are conflicting. Now in transactions, if there is conflicting operations and there is a way this conflicting operations have been executed, let us say the conflicting operation in this particular case is executed is such a way that this is the order in a schedule. Now this order actually determines that T_1 actually preceded T_2 because it is conflicting on data item x and T_1 has been executed before T_2.

[Refer Slide Time: 17.02]

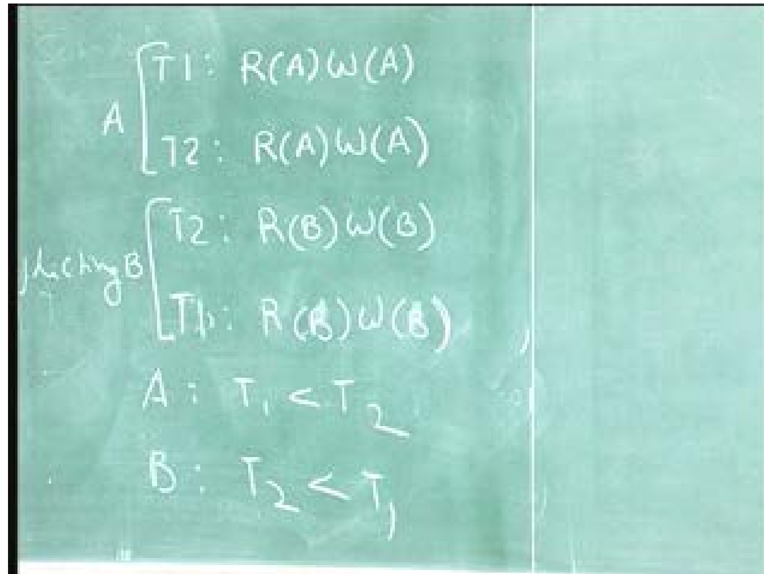


And now this order should be preserved, no matter what happens with respect to other operations. And as long as you preserve that order with respect to all other operations, we say that the operations have been executed in a serial fashion or the schedule is reducible to a serial schedule. This is the concept of serializibilty. Now this is the important notion here is when transactions are executing concurrently. We need to ensure that the conflicting operations are serializable, all the conflicting operations are serializable.

Now here is a very simple case shown in the slide where it shows that where it is not possible to serialize, we need to actually abort the transaction. Now in this particular case it is shown that T_1 actually is reading as we go to the earlier case, T_1 is actually reading the data item A and then writing data item A. Now if you look at the T_2 is also reading data item A and then writing data item A. Now the other part of T_2 is read B and write B. Now as you can see here with respect to data item A, the order between T_1 and T_2 is T_1 is before T_2.

Now if you look at the data item B, it is coming in the revere direction which actually means that as you can see here, on the data item B as far as the conflicting data item A is concerned, T_1 is before T_2, as far as B is concerned it is T_2 before T_1. As you can see this is on data item B and this is on data item A.

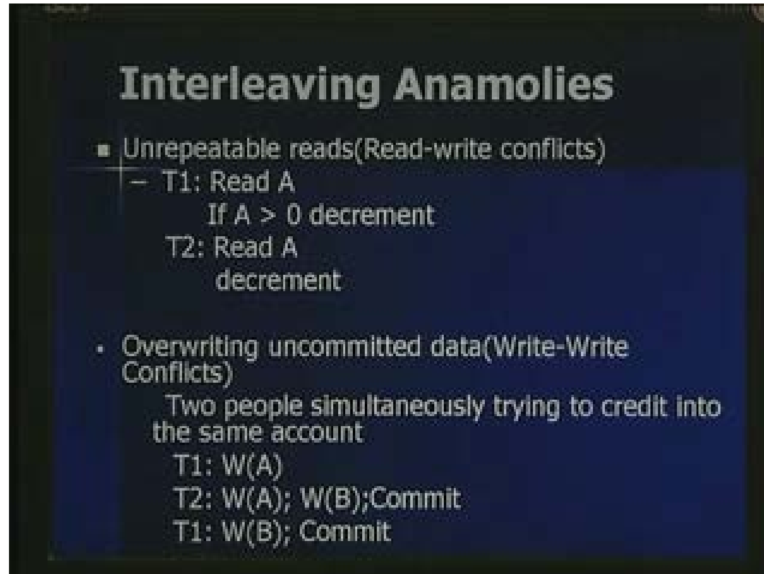
[Refer Slide Time: 19.18]



Now from this it is not possible for us to say whether T_1 actually has finished before T_2 or T_2 has finished before T_1. Since we can't now decipher which one has actually finishing before the other, this schedule is non serializable schedule. And this is what should be avoided. A non serialzable schedule shows that the execution of the operations will lead to inconsistency. The database will be in an inconsistent state when we have the operations executed in a non serlizable way.

Now in this particular case it is shown here that T_2 can commit but T_1 has to abort. That's what was shown in the slide saying that only B can come, transaction T_2 can commit but T_1 has to abort because T_1 trying to commit here will produce a non serialzable schedule. This is also shown, as if you go back to the slide you can see that this is shown as dirty reads, that is write and read conflicts. As you can see B has been, T_1 has read A and B values but it is reading the, writing the value at a much later stage. And now the one of the later updates will be lost if you allow T_1 and T_2 to execute in this particular way. One of the operations will be lost and that's the reason why the schedule is not allowed. We look at other kinds of conflicts that can arise when transactions are executing. First kind of conflict that we saw is a write read conflict.

[Refer Slide Time: 21.08]



We can also have a read write conflict and a write write conflict. what a read write conflict shows is the example shown here is transaction T_1 is reading the data item A and if its value is more than zero then it decrements. T_2 is actually reading A and it is decrementing the value of A. Now when T_1 and T_2 are operating concurrently, T_1 is actually reading data item whereas T_2 is actually writing on to the data item. This is what we mean by a read write conflict on the data item A, with respect to data item A both transaction T_1 and T_2 are conflicting in terms of T_1 reading the data item and T_2 modifying the data item. And this is what we mean by a read write conflict.

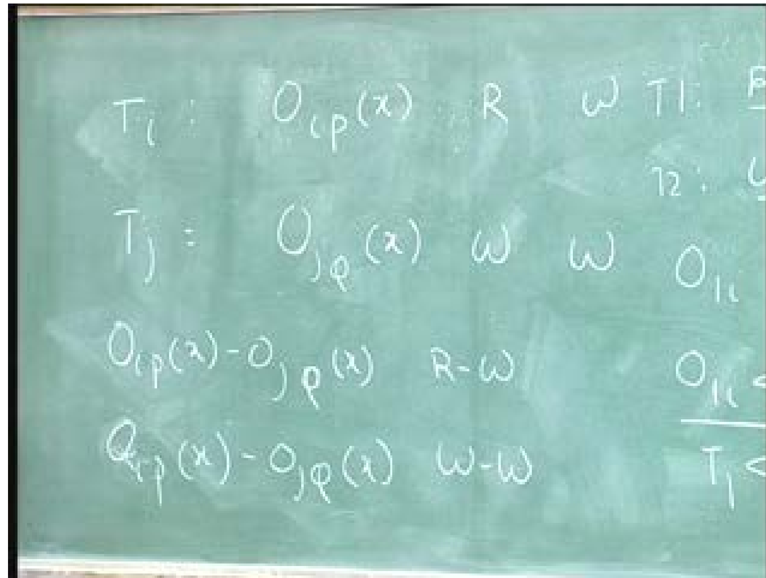
Again you can see that in this particular case the value read by T_1, if it is before the value is changed will be inconsistent. Let us say right now the value of A is zero and T_1 reads it as zero, then it is unlikely to decrement that because it has read the value as zero. Now if T_2 also read the value at the same time as zero and reads it as zero then it decrements it as minus 1. So that value is actually in conflict. The actual value that is read by T_1 and T_2 are not correct.

Another example for conflict is the write write conflict where two transactions, both of them access the data items and tries to modify the value of the data item. This is a case where it is shown here as two people simultaneously trying to credit into the same account is shown as a example of a write write conflict. As can be seen here, T one is actually writing the value into the account, T two is also writing a value in the account A as well as into B and T_1 at a later stage is trying to write the value into B. Now as you can see in this particular case, the writes on the same data item will be conflicting leading to non serializable schedule. So typically this is what we mean by a write write conflict.

To summarize what we are actually seeing, I will give a simple example of an operation and show how exactly is the conflicts serializability is to be achieved. If you take a transaction T_i and say T_i as a operation o_{ip} and data item x and you have a transaction T j

which has an operation q on x. now we say that these operations are conflicting if one them is write. Now when you say this is write and this is a read, you have between o_{ip} x and $o_{j\,q}$ you basically have a read write conflict. The x data item is read by transaction T i and it is been modified by T j. So this is basically a read write conflict. Now if you say that this is a write write, typically what you see is between operation o i p x and o j q x you see a write write conflict.

[Refer Slide Time: 25.01]

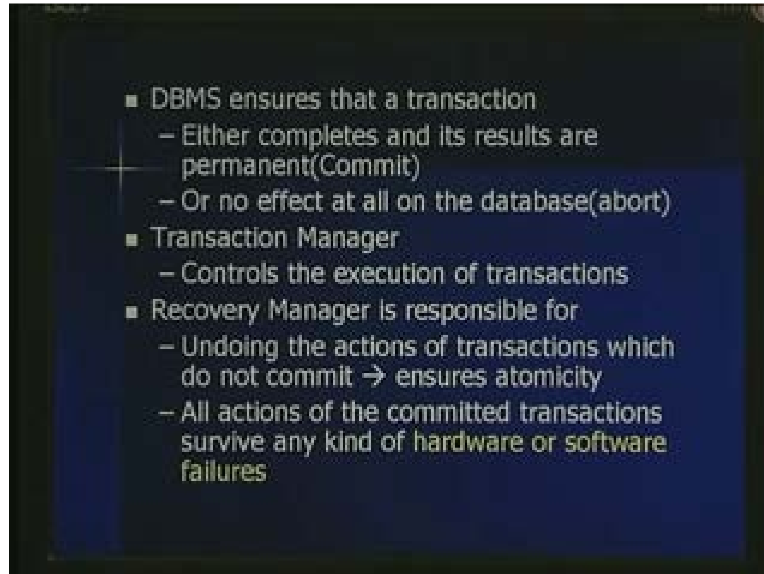


Now when there is a conflict between the two operations, whether it is a read write or a write write conflict, you need to actually serialize the operations by actually saying that they are executed in a serial order which is what we mean by conflict serializablitiy. Now whenever there is a conflicting operations, we need to actually serialize the two operations which is known as the conflict serializability.

In this particular case $o_{i\,p}$ (x) and $o_{j\,q}$ (x) are the conflicting operations and they need to be serialized in a particular fashion and this is what we mean by conflict serializability. We are going to see in later lectures, how the transactions are executed by the transaction manager to ensure that conflicting operations are serialized or serial schedules are produced by the transaction manager.

One of the simple technique that is used is what we see as a two phase locking and we are going to study that two phase locking as a technique for achieving conflict serailizability later in our lectures. Now here is actually what is shown as how exactly the transaction manager achieves some of the properties that we have been discussing.

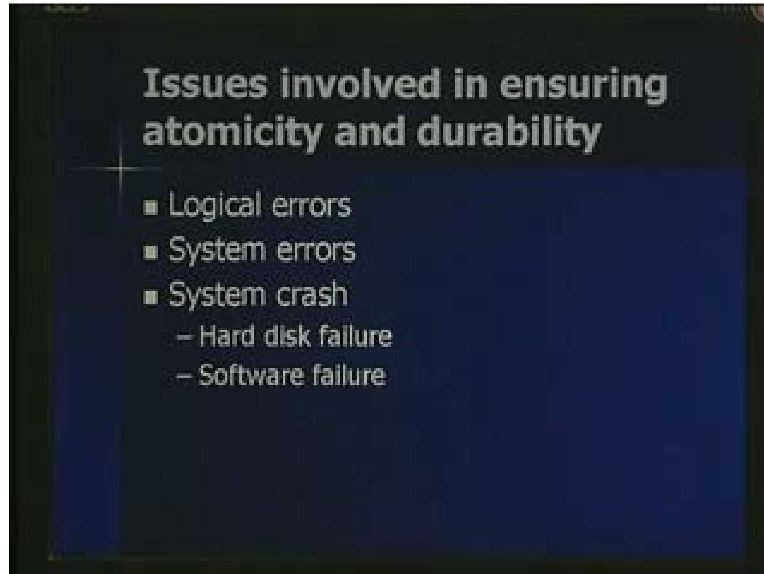
[Refer Slide Time: 26.52]



The dbms ensure that a transaction either completes and its results are permanently written. This is what we mean by committing a transaction or no effect at all on the database, this is equivalent to saying that the transaction has been aborted. So we have two states for the transaction, either a commit state or an abort state. In the case of a commit state, all the operations of the transaction or executed in full and then they are committed. In the case of abort, no effect at all on the database as far as that transaction is concerned. Now the idea of transaction manager is it controls the execution of the transactions. As we saw in this particular case, it controls the execution of the transactions in such a way that the operations of the transactions are serializable.

Similarly if you take the recovery manager, recovery manager is responsible for undoing the actions of transaction which do not commit. This is a equivalent to saying that the recovery manger is responsible for ensuring the property of atomicity. All actions of the committed transactions survive any kind of hardware or software failures. This is actually known as writing the committed transactions on to a stable storage. What we are going to do is we are going to look at little further into how the recovery manager ensures that properties of atomicity and durability.

[Refer Slide Time: 28.28]



Now what are issues involved in ensuring atomicity and durability? The following errors can occur when a transaction is executing. First is it could relate to logical errors. For example you are trying to withdraw some money from a bank account, it is possible that the account itself doesn't exist or the account doesn't has sufficient funds. In all this cases, transaction cannot proceed any further. This is what we mean by logical errors. The transaction may have to abort because of logical errors. There could be system errors. For example it is possible that there are problems of network, there are problems of system failures, temporary failure or power failure in which case when the power comes back, you need to know what really happened for your transaction with respect to already started transactions.

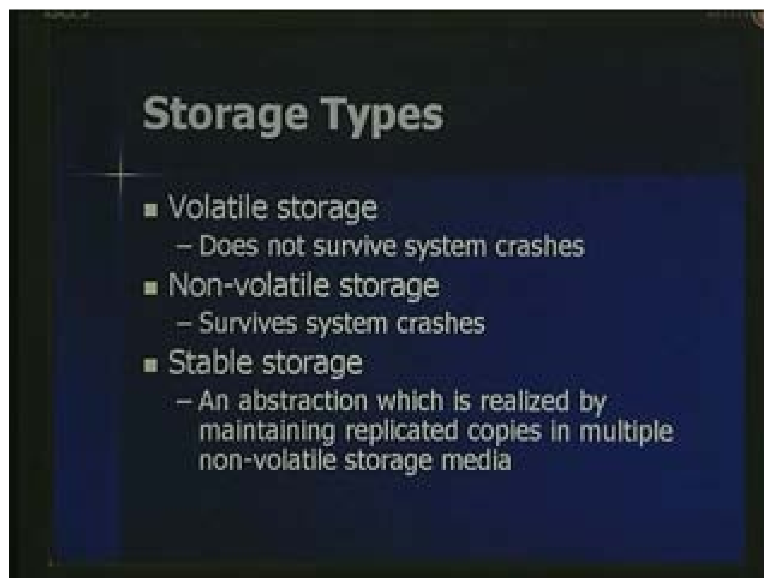
A simple example could be, you go to a atm and try to withdraw money from the atm and the power, when you actually press the button for withdrawing the money, the atm stops functioning. atm failed due to various reasons. You would like to know whether their system is actually debited the amount from the bank, from your balance or not. That is basically system errors. There could be crash, system crashes, there could be a hard disk failure, the disk head could have been corrupted. So there could be various reasons why the system didn't' perform, it could be a system crash. So all these errors are possible.

When the system actually goes into any of these errors, you want to understand how exactly the atomicity and the durability properties can be maintained. A simple example trying to illustrate this point will be something like a file which all of us open on a windows machine or any of our unix machines. Now here when you open a file in a editor mode and tries to edit your file, there is no guarantee in terms of what happens when a power fails because the file could be in a very corrupted state. there is no guarantee for you in terms of the state of the file which all of us know we keep repeatedly saving the file, when we actually entering or writing some document, we try actually

saving the document as many times as possible, so that when the power goes off or something else happens we still save the portion of the work we have actually done.

We don't lose the file because of power failure all the work that we have done. Now the same thing cannot happen in the case of database systems because here the more critical data that is been in saved in the file. So we need to ensure that whatever happens when any of these failures happen, the system is still in a predictable state. That is the difference between ordinary file systems implemented by an operating system and a database implemented by commercial systems. They ensure that whenever these things happen, still the properties, the save properties for the transaction, the acid properties of the transactions are retained.

[Refer Slide Time: 32.11]

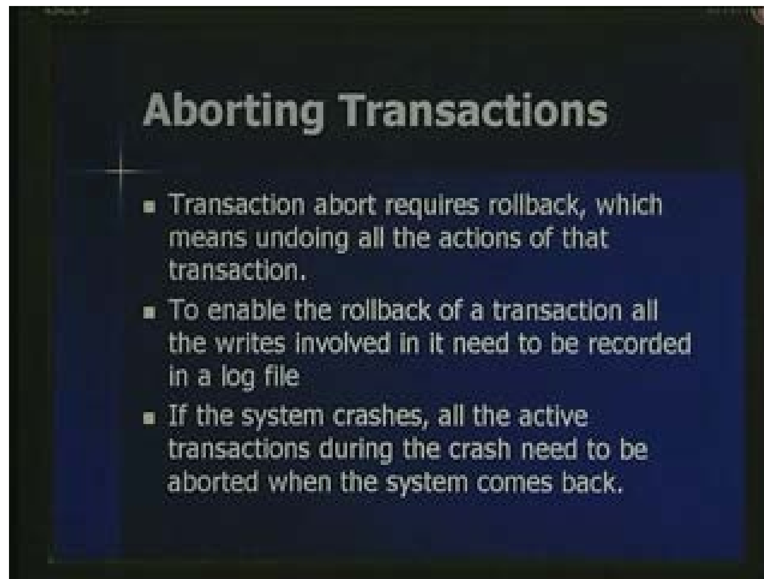


To explain how these properties are retained by the system, we need to also understand the different storage types that are available in a computer system. A simple volatile storage, we basically look at a simple volatile storage, this does not survive system crashes that means when the system actually crashes the storage is lost, the storage is volatile it is lost the minute the system crashes. When you talk about non volatile storage, the system actually survives these crashes. That means the storage is the, whatever you write into the storage is not lost when the system crash occurs.

A simple case is whatever is there in the main memory is lost when the power goes whereas if you have written it onto your hard disk, it survives a power failure because it is written into a more non volatile storage. Now we also have a concept of a stable storage which is an abstraction of maintaining replicated copies in multiple non volatile storage media, so that whenever higher disasters occur we still have a way of getting our data back and that is we mean by the concept of a stable storage.

Now what we are going to see is how this concepts are used for actually achieving the atomicity and durability properties in the transaction manager.

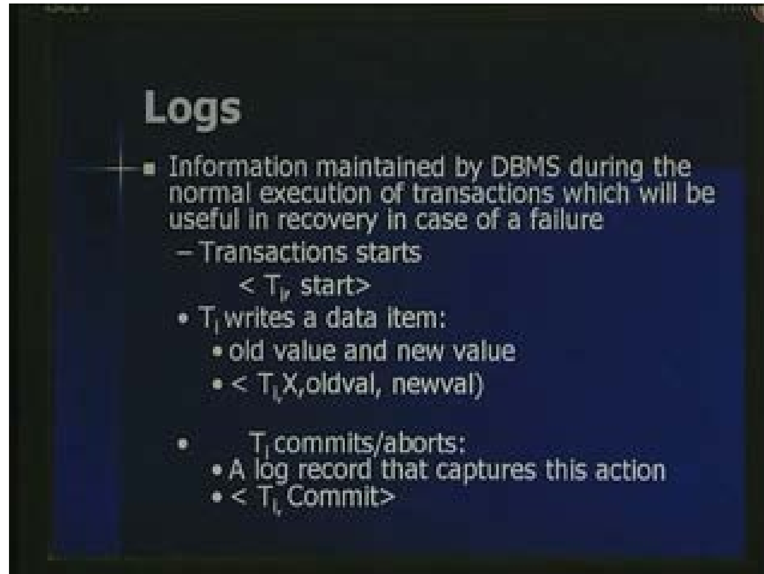
[Refer Slide Time: 33.55]



What we are going to show here in this particular case is what really happens when transactions have to roll back. The rolling of the transaction has to happen mainly because of logical errors or the system crashes and hence it has to be restored back to a previous state. Transaction abort requires roll back which means undoing all the actions of that particular transaction.

Now to ensure that roll back of the transaction occurs properly, what we have to do is all the writes of the transaction have to be properly recorded in what is called as a log file. The log file retains all the information relating to the writes of the transaction and this will be used when the roll back has to occur. Now if the system crashes, all active transactions during the crash need to be aborted when the system comes back. This is equivalent to saying that they will all be rolled back and the information that is there in the system in the log file will be used to properly undo the transaction activities, whatever the transactions are being doing. What we are going to do is again in this particular case, we will take a very simple example and see how exactly this happens.

[Refer Slide Time: 35.22]

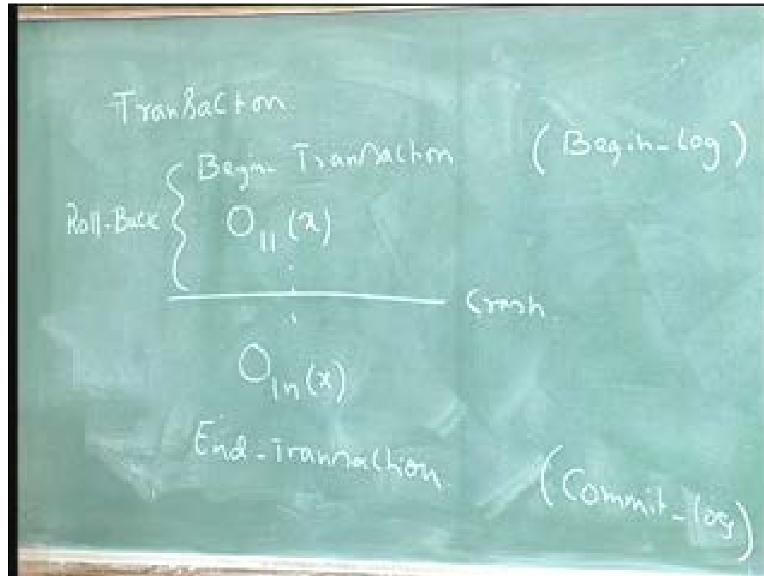


Now here is the case where case where the logs are maintained and how this logs are helpful in ensuring the atomicity property or the how the atomicity property will be realized by the database manager. What is shown here is a simple case of writing the logs before the transactions starts executing and making sure that it is carried over whenever the data item is being written or a new value for the data item is being written.

What you can see here is the first log that will be written as far as the transaction T one is concerned is what is called is the begin log, transacting begin log. Now what we have is basically a relating to a transaction, every transaction is preceded by the begin transaction as a keyword. Now this begin transaction actually tells the database manager to write what we see as a log, this is the transaction begin log.

Now this is a, the begin log has to be written onto the transaction. Now in between the transaction does various operations. As we have seen their, it's possible that there are several operations which are done by the transaction in between and then we have and end transaction. Now this is actually the last instruction that is executed by this transaction, so we will have a what is called a commit log indicating that the transaction has actually committed which is equivalent to saying that all these operation have been successfully executed. So between begin and end, at any point of time when there is a crash, we need to actually recover back to the starting point. And this is what we mean by actually roll back. What we are calling as a roll back is basically rolling back all the things that a transaction is done to the beginning. This is what is meant as a roll back. Now what we do is we actually ensure that whatever the transaction is doing, is written on to this log and this log will be used for rolling back the transaction whenever a crash occurs.

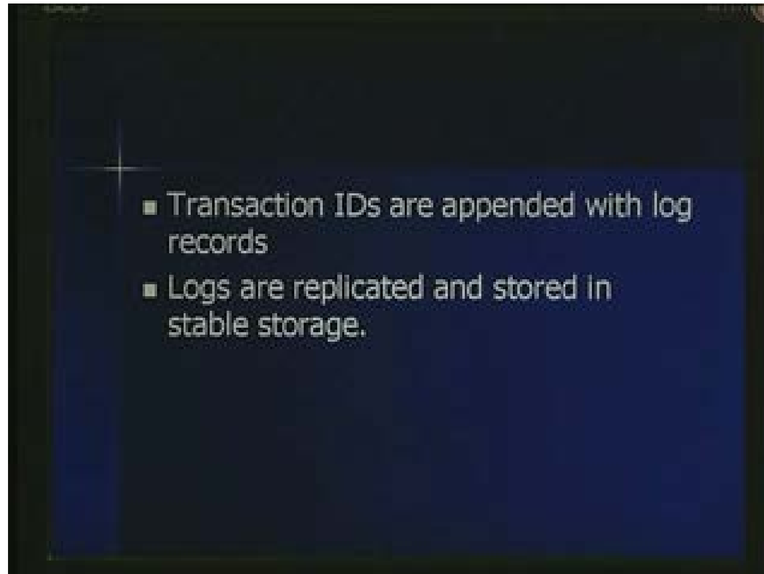
[Refer Slide Time: 37.57]



For example you can see that in the slide it is shown that T i writes a data item. Now as you can see the first one transaction starts, there is a transaction log T i start. Now the second thing that you notice is T i writes a data item. now there is a old value and a new value old value is the old value of the data item and the new value is the new value of the data item x. so there is a log that is written there which shows that T i x old value and a new value is shown here, this is how actually the log is written.

Whenever there is a change in the data item, we basically write the log and now this log shows what was the old value and what is the new value now. When you come to the last transaction, basically you have a commit log that is a T i commit log. So as shown here, we have a begin log and a commit log and in between whatever is happening is being recorded there as shown in the slide there. So one of the things that we are going to look at now is how this logs can be used for recovery purposes. How exactly this logs can be used by the database manager to ensure that whenever those kind of failures that we are talking earlier occurs how the system will recover back from those failures.

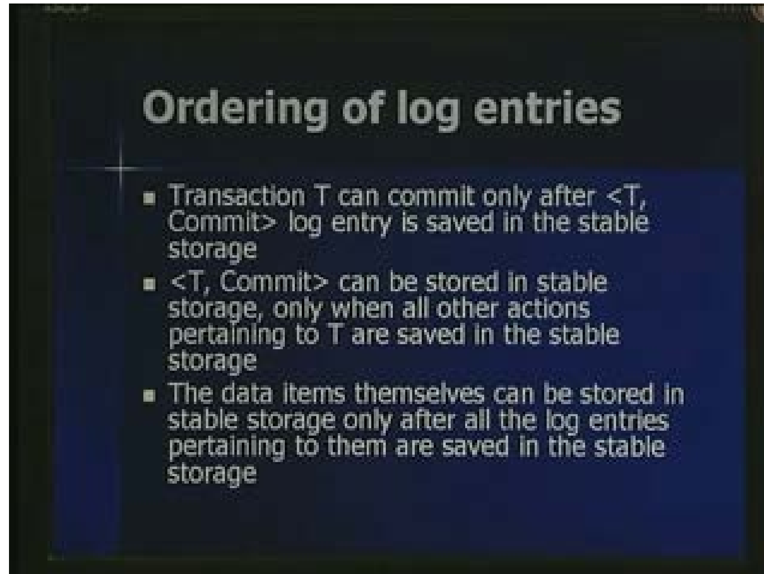
[Refer Slide Time: 39.54]



Now one of the things when writing this logs is one of the things that one should remember is the id's of the transactions are stored. So that we know to this logs pertain to which transactions, so transaction id's are appended when the logs records to identify the transactions for which this have logs been produced. Logs are replicated and stored in a stable storage. This is also very important because in the logs themselves are lost then there is no way you can recover back.

Logs only assuming that the logs are written on to a stable storage, you can ensure that the transaction can be made to recover. But if the logs themselves are subjected to failure then you will not be able to recover back and hence logs are replicated, one of the assumption we make it logs are replicated and they are stored in a stable storage. So when we say a log is written, we assume that the values relating to the log have been written on a stable storage and it is possible for us to recover this information at any point of time.

[Refer Slide Time: 41.10]



Now as you can see here we are also showing how this log entries will also be ordered, ordering of the log entries. We say a transaction T are can commit only if the log entry relating to that is saved on a stable storage. This is equivalent to saying as you can see here, when you write this commit log this commit log is actually written on to a stable storage then we say the transaction is committed. This is the point where it is possible for the transaction now to say that it is committed. Now before this is actually written, this log is written all the other entries before this pertaining to this transaction should have also been written onto the stable storage.

You should never write the commit log before all the other log entries relating to this transaction have been saved on the stable storage. Now only after the writing the entries relating to the logs, you should write the data items themselves after this point onto the stable storage. This is very important, these steps are very important because if you perform them in any other order, you will have problems in terms of recovering back. First requirement is all the log entries relating to this transaction should have written onto the stable storage in the first instance, before you are writing the commit log. Only after writing the commit log, the data item values pertaining to the transaction can themselves can be written onto the stable storage.

The reason for this is simple. If you don't write the log values first on to the stable storage, there is no way if something happens to recover from that particular failure. for example if you have written the data value onto the stable storage, now something happens there is now way of finding out what is the state in which the transaction is unless the logs are written properly. So logs are the bases for the database manager to find out what is the state in which the transaction is when a failure is occurred. And hence it is important for you to first write all the logs relating to the transaction then write the commit log and then write all the data items onto the stable storage. This is how one needs to order or write the various things relating to the transaction.

[Refer Slide Time: 44.09]

Example

- T1:
  Read(A)
  A = A + 50
  Read(B)
  B = B+100
  Write(B)
  Read(C)
  C = 2C
  Write(C)
  A = A+B+C
  Write(A)

T2:
  Read(A)
  A = A+10
  Read(D)
  D = D − 10
  Read(E)
  Read(B)
  E = (E + B)
  Write(E)
  D = D + E
  Write(E)

Initial values are
A: 100 B:300 C: 5 D:60 E:80

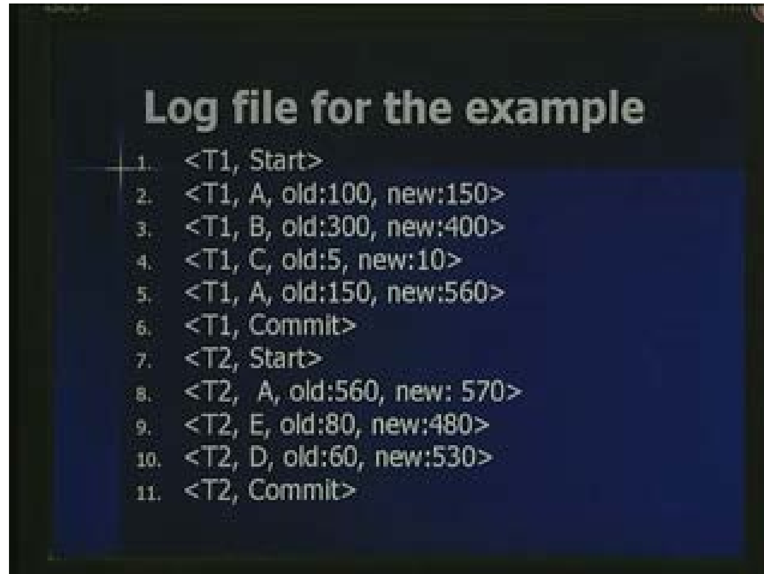
We will take a very simple example. A simple example in this case to see how the two transactions can really execute writing their logs. In this particular case, it is shown T one and T two T one is actually reading certain data items and writing certain data items. Similarly T two is also reading and writing certain data items. For completeness sake, we also have shown the initial values that are there in the database when this transactions T one and T two start executing.

As can be seen in the slide, the initial values of A are A is 100, B is 300, c is 5, D is 60 and E is 80. Now T_1 when starts executing, it is going to read the values of A will increment by 50 then read the value of B increment it by 100 then write the value of B back into the system. Then will read the value of C. Now C value is inc is double then the value of C is again written. At the end of it A is recomputed as A plus B plus C and the value of A is written.

If you look at the transaction T_2, T_2 is actually reading the value of A. It is incrementing it by 10 then it is actually reading the value of D, it is decrementing it by 10 then actually reading the value of E and reading the value of B. Then E value is computed by adding E plus B and then writing the value of E then D is recomputed as D plus E then finally the value of E is written.

What we wish to covey with this is the two transactions are simultaneously reading several data items and trying to modify. It is equivalent to saying that there is set of operations which are going simultaneously in terms of reads and writes between these two transactions.

[Refer Slide Time: 46.21]

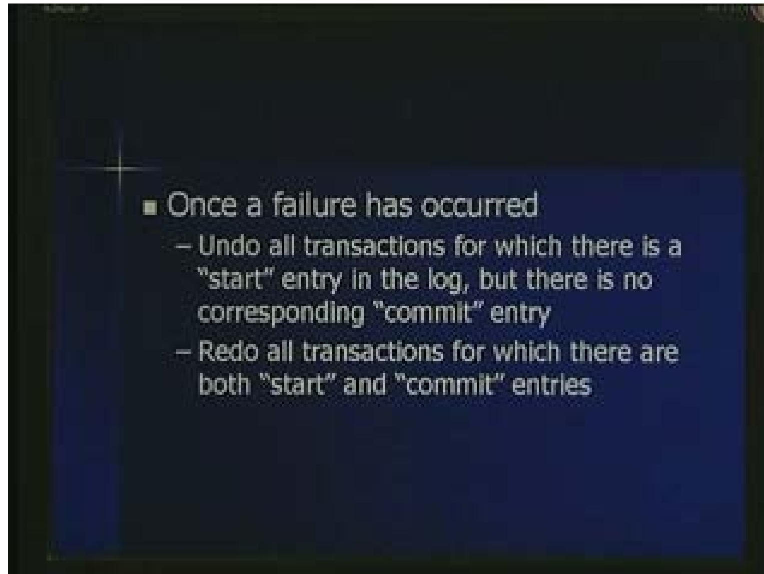


Now let us understand what really would have been the logs that would have been produced when this transaction T_1 and T_2 start executing. Now when T_1 starts executing you can see that there is an initial log given there as T_1 start. This actually shows that T one started executing that is the start log for the transaction T. T_1 is the id of the transaction. Now when actually T_1 tried writing the value of A, the old vale and the new value are actually stored in the database, in the log.

As you can see in this particular case, this log shows that A's old value is 100 and the new computed value is 150. Similarly when B has actually been recomputed, we have a old value for B as 300 and the new value has 400. Similarly for the value of C, it is 5 and 10 and finally when the last computation for A actually took place. A is 150 and then old value of A is 150 and then the new value is 560. It is at this point of time actually the T_1 finished executing all its instructions and it is ready for commit.

And that is the time the commit log is written, T_1 commit log is produced at that point of time. A similar thing is shown for T_2, as you can see there is a start log for T_2 and then we have various new values and old values for the data item that are done by T_2 are also shown here and finally commit log is shown for T_2.

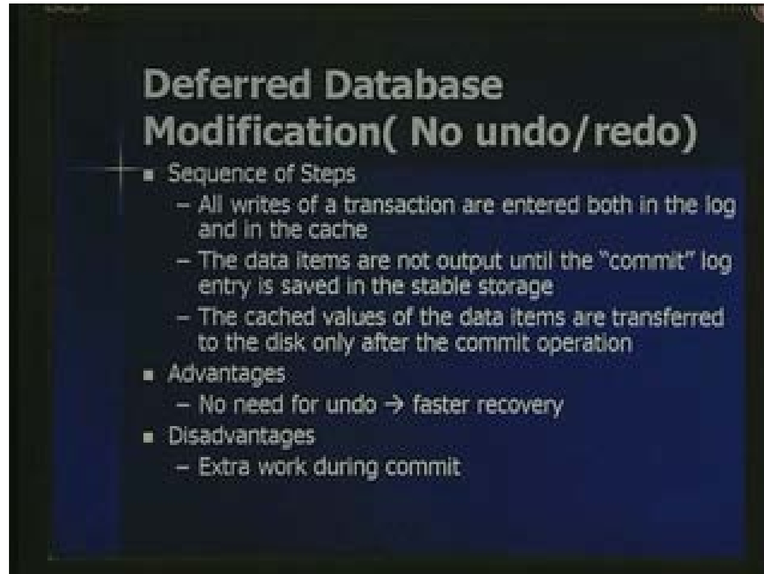
[Refer Slide Time: 48:08]



Now what really happens is once a failure is occurred, undo all transactions for which there is a start entry in the log but there is no corresponding commit entry. This is equivalent to saying that I will just go back to the previous slide to show what really we are we are saying. Now we can say if T when you actually a system crash has occurred, you can see there is a start log for T_1 but corresponding commit log is not present. Then all that we have to do is you have to actually undo whatever has been done by the transaction.

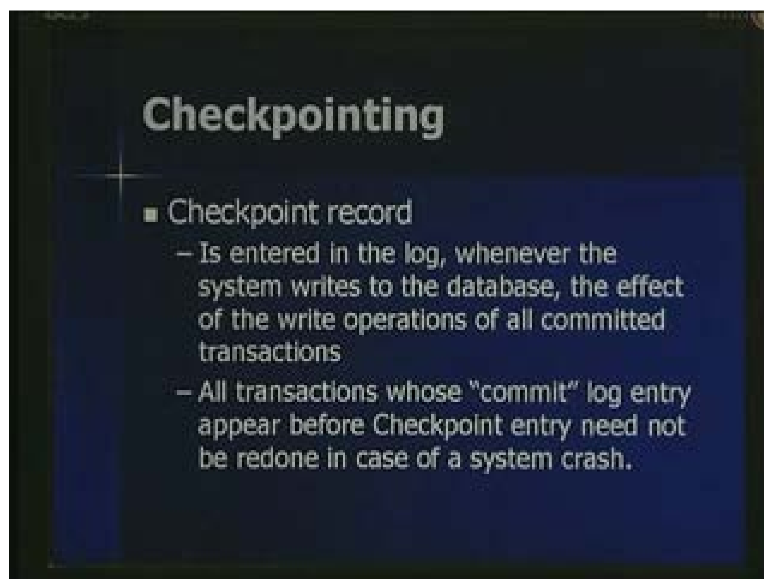
For example in this particular case, if T_1 has actually modified the values of A B C then those values have to be reset back to the old values from the new values. That is equivalent to actually saying that we have undone the transaction because it has not reached the commit state. Now the other case is redo all transactions for which there are both start and commit entries because these transactions have already gone to the finish stage. We are going to redo the transaction for all those which commit, a start and the commit entries are there.

[Refer Slide Time: 49.26]



And also if typically we have, if the data items are not yet written this is what we mean by actually looking at whether the, not just the log entries but the data item entries are not still written then we need to redo. But if the data item entries are also been stored on the stable storage, there would have been along with commit log there would have been a complete log and the log book showing that the transaction has completed. All the operations relating to it, in which case the redo need not be done and this is typically achieved by what is called a checkpoint record.

[Refer Slide Time: 50.04]



If the checkpoint record is entered into the log, whenever the system writes the data item values onto the database. The effect of the write operations are all committed on the transactions. Now all transactions whose commit log entry appears before the check point entry need not be redone in case of a system crash. So the checkpoint is a place where you can decide whether when a commit log exists whether you have redo or need not redo those transactions. This is how exactly the database manager will ensure, the transactions are executed atomically and they satisfy the property of durability. What we are going to see in the next class is how the concurrency control properties of the transactions are realized by the transaction manager.