# Transaction Management in DBMS

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

A transaction is a set of logically related operations. For example, you are transferring money from your bank account to your friend’s account, the set of operations would be like this:

## Simple Transaction Example

1. Read your account balance  
2. Deduct the amount from your balance  
3. Write the remaining balance to your account  
4. Read your friend’s account balance  
5. Add the amount to his account balance  
6. Write the new updated balance to his account

This whole set of operations can be called a transaction. Although I have shown you read, write and update operations in the above example but the transaction can have operations like read, write, insert, update, delete.

In DBMS, we write the above 6 steps transaction like this:  
Lets say your account is A and your friend’s account is B, you are transferring 10000 from A to B, the steps of the transaction are:

000

1. R(A);

2. A = A - 10000;

3. W(A);

4. R(B);

5. B = B + 10000;

6. W(B);

In the above transaction R refers to the Read operation and W refers to the write operation.

## Transaction failure in between the operations

Now that we understand what is transaction, we should understand what are the problems associated with it.

The main problem that can happen during a transaction is that the transaction can fail before finishing the all the operations in the set. This can happen due to power failure, system crash etc. This is a serious problem that can leave database in an inconsistent state. Assume that transaction fail after third operation (see the example above) then the amount would be deducted from your account but your friend will not receive it.

To solve this problem, we have the following two operations

Commit: If all the operations in a transaction are completed successfully then commit those changes to the database permanently.

Rollback: If any of the operation fails then rollback all the changes done by previous operations.

Even though these operations can help us avoiding several issues that may arise during transaction but they are not sufficient when two transactions are running concurrently. To handle those problems we need to understand database [**ACID properties**](https://beginnersbook.com/2015/04/acid-properties-in-dbms/).

# ACID properties in DBMS

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

To ensure the integrity of data during a transaction (A transaction is a unit of program that updates various data items, read more about it here), the database system maintains the following properties. These properties are widely known as ACID properties:

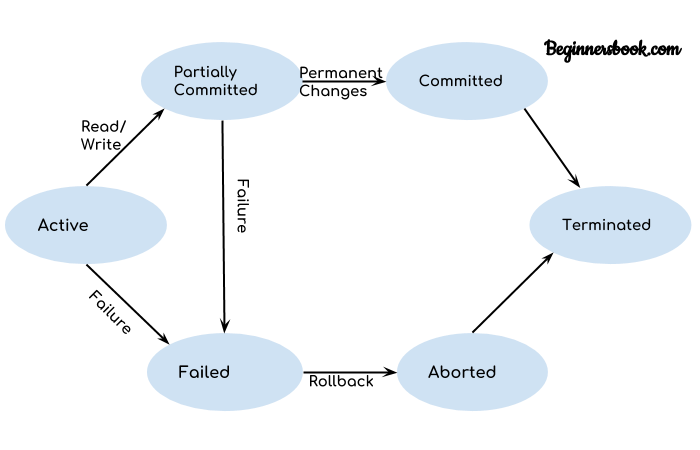
* Atomicity: This property ensures that either all the operations of a transaction reflect in database or none. Let’s take an example of banking system to understand this: Suppose Account A has a balance of 400$ & B has 700$. Account A is transferring 100$ to Account B. This is a transaction that has two operations a) Debiting 100$ from A’s balance b) Creating 100$ to B’s balance. Let’s say first operation passed successfully while second failed, in this case A’s balance would be 300$ while B would be having 700$ instead of 800$. This is unacceptable in a banking system. Either the transaction should fail without executing any of the operation or it should process both the operations. The Atomicity property ensures that.
* Consistency: To preserve the consistency of database, the execution of transaction should take place in isolation (that means no other transaction should run concurrently when there is a transaction already running). For example account A is having a balance of 400$ and it is transferring 100$ to account B & C both. So we have two transactions here. Let’s say these transactions run concurrently and both the transactions read 400$ balance, in that case the final balance of A would be 300$ instead of 200$. This is wrong. If the transaction were to run in isolation then the second transaction would have read the correct balance 300$ (before debiting 100$) once the first transaction went successful.
* Isolation: For every pair of transactions, one transaction should start execution only when the other finished execution. I have already discussed the example of Isolation in the Consistency property above.
* Durability: Once a transaction completes successfully, the changes it has made into the database should be permanent even if there is a system failure. The recovery-management component of database systems ensures the durability of transaction.

# DBMS Transaction States

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

In this guide, we will discuss the states of a transaction in DBMS. A transaction in DBMS can be in one of the following states.

## DBMS Transaction States Diagram



Lets discuss these states one by one.

## Active State

As we have discussed in the [**DBMS transaction introduction**](https://beginnersbook.com/2017/09/transaction-management-in-dbms/) that a transaction is a sequence of operations. If a transaction is in execution then it is said to be in active state. It doesn’t matter which step is in execution, until unless the transaction is executing, it remains in active state.

## Failed State

If a transaction is executing and a failure occurs, either a hardware failure or a software failure then the transaction goes into failed state from the active state.

## Partially Committed State

As we can see in the above diagram that a transaction goes into “partially committed” state from the active state when there are read and write operations present in the transaction.

A transaction contains number of read and write operations. Once the whole transaction is successfully executed, the transaction goes into partially committed state where we have all the read and write operations performed on the main memory (local memory) instead of the actual database.

The reason why we have this state is because a transaction can fail during execution so if we are making the changes in the actual database instead of local memory, database may be left in an inconsistent state in case of any failure. This state helps us to rollback the changes made to the database in case of a failure during execution.

## Committed State

If a transaction completes the execution successfully then all the changes made in the local memory during partially committed state are permanently stored in the database. You can also see in the above diagram that a transaction goes from partially committed state to committed state when everything is successful.

## Aborted State

As we have seen above, if a transaction fails during execution then the transaction goes into a failed state. The changes made into the local memory (or buffer) are rolled back to the previous consistent state and the transaction goes into aborted state from the failed state. Refer the diagram to see the interaction between failed and aborted state.

# DBMS Schedules and the Types of Schedules

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

We know that [**transactions**](https://beginnersbook.com/2017/09/transaction-management-in-dbms/) are set of instructions and these instructions perform operations on database. When multiple transactions are running concurrently then there needs to be a sequence in which the operations are performed because at a time only one operation can be performed on the database. This sequence of operations is known as Schedule.

Lets take an example to understand what is a schedule in DBMS.

## DBMS Schedule example

The following sequence of operations is a schedule. Here we have two transactions T1 & T2 which are running concurrently.

This schedule determines the exact order of operations that are going to be performed on database. In this example, all the instructions of transaction T1 are executed before the instructions of transaction T2, however this is not always necessary and we can have various types of schedules which we will discuss in this article.

T1 T2

---- ----

R(X)

W(X)

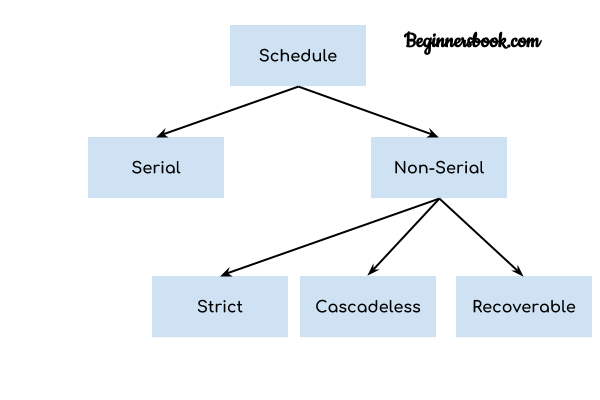
R(Y)

R(Y)

R(X)

W(Y)

## Types of Schedules in DBMS

We have various types of schedules in DBMS. Lets discuss them one by one.  


### Serial Schedule

In Serial schedule, a transaction is executed completely before starting the execution of another transaction. In other words, you can say that in serial schedule, a transaction does not start execution until the currently running transaction finished execution. This type of execution of transaction is also known as non-interleaved execution. The example we have seen above is the serial schedule.

Lets take another example.

Serial Schedule example  
Here R refers to the read operation and W refers to the write operation. In this example, the transaction T2 does not start execution until the transaction T1 is finished.

T1 T2

---- ----

R(A)

R(B)

W(A)

commit

R(B)

R(A)

W(B)

commit

### Strict Schedule

In Strict schedule, if the write operation of a transaction precedes a conflicting operation (Read or Write operation) of another transaction then the commit or abort operation of such transaction should also precede the conflicting operation of other transaction.

Lets take an example.

Strict Schedule example  
Lets say we have two transactions Ta and Tb. The write operation of transaction Ta precedes the read or write operation of transaction Tb, so the commit or abort operation of transaction Ta should also precede the read or write of Tb.

Ta Tb

----- -----

R(X)

R(X)

W(X)

commit

W(X)

R(X)

commit

Here the write operation W(X) of Ta precedes the conflicting operation (Read or Write operation) of Tb so the conflicting operation of Tb had to wait the commit operation of Ta.

### Cascadeless Schedule

In Cascadeless Schedule, if a transaction is going to perform read operation on a value, it has to wait until the transaction who is performing write on that value commits.

Cascadeless Schedule example  
For example, lets say we have two transactions Ta and Tb. Tb is going to read the value X after the W(X) of Ta then Tb has to wait for the commit operation of transaction Ta before it reads the X.

Ta Tb

----- -----

R(X)

W(X)

W(X)

commit

R(X)

W(X)

commit

### Recoverable Schedule

In Recoverable schedule, if a transaction is reading a value which has been updated by some other transaction then this transaction can commit only after the commit of other transaction which is updating value.

Recoverable Schedule example  
Here Tb is performing read operation on X after the Ta has made changes in X using W(X) so Tb can only commit after the commit operation of Ta.

Ta Tb

----- -----

R(X)

W(X)

R(X)

W(X)

R(X)

commit

commit

# DBMS Serializability

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state. Serializability is a concept that helps us to check which [**schedules**](https://beginnersbook.com/2018/12/dbms-schedules/) are serializable. A serializable schedule is the one that always leaves the database in consistent state.

## What is a serializable schedule?

A serializable schedule always leaves the database in consistent state. A [**serial schedule**](https://beginnersbook.com/2018/12/dbms-schedules/) is always a serializable schedule because in serial schedule, a transaction only starts when the other transaction finished execution. However a non-serial schedule needs to be checked for Serializability.

A non-serial schedule of n number of transactions is said to be serializable schedule, if it is equivalent to the serial schedule of those n transactions. A serial schedule doesn’t allow concurrency, only one transaction executes at a time and the other starts when the already running transaction finished.

## Types of Serializability

There are two types of Serializability.

# DBMS Conflict Serializability

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

In the [**DBMS Schedules**](https://beginnersbook.com/2018/12/dbms-schedules/) guide, we learned that there are two types of schedules – Serial & Non-Serial. A Serial schedule doesn’t support concurrent execution of transactions while a non-serial schedule supports concurrency. We also learned in **[Serializability](https://beginnersbook.com/2018/12/dbms-serializability/)** tutorial that a non-serial schedule may leave the database in inconsistent state so we need to check these non-serial schedules for the Serializability.

Conflict Serializability is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.

## What is Conflict Serializability?

A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

### Conflicting operations

Two operations are said to be in conflict, if they satisfy all the following three conditions:

1. Both the operations should belong to different transactions.  
2. Both the operations are working on same data item.  
3. At least one of the operation is a write operation.

Lets see some examples to understand this:  
Example 1: Operation W(X) of transaction T1 and operation R(X) of transaction T2 are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operation in write operation.

Example 2: Similarly Operations W(X) of T1 and W(X) of T2 are conflicting operations.

Example 3: Operations W(X) of T1 and W(Y) of T2 are non-conflicting operations because both the write operations are not working on same data item so these operations don’t satisfy the second condition.

Example 4: Similarly R(X) of T1 and R(X) of T2 are non-conflicting operations because none of them is write operation.

Example 5: Similarly W(X) of T1 and R(X) of T1 are non-conflicting operations because both the operations belong to same transaction T1.

## Conflict Equivalent Schedules

Two schedules are said to be conflict Equivalent if one schedule can be converted into other schedule after swapping non-conflicting operations.

## Conflict Serializable check

Lets check whether a schedule is conflict serializable or not. If a schedule is conflict Equivalent to its serial schedule then it is called Conflict Serializable schedule. Lets take few examples of schedules.

### Example of Conflict Serializability

Lets consider this schedule:

T1 T2

----- ------

R(A)

R(B)

R(A)

R(B)

W(B)

W(A)

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is not Conflict Serializable.

Lets take another example:

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

Lets swap non-conflicting operations:

After swapping R(A) of T1 and R(A) of T2 we get:

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

After swapping R(A) of T1 and R(B) of T2 we get:

T1 T2

----- ------

R(A)

R(B)

R(A)

W(B)

R(B)

W(A)

After swapping R(A) of T1 and W(B) of T2 we get:

T1 T2

----- ------

R(A)

R(B)

W(B)

R(A)

R(B)

W(A)

We finally got a serial schedule after swapping all the non-conflicting operations so we can say that the given schedule is Conflict Serializable.

# DBMS View Serializability

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

In the last tutorial, we learned [**Conflict Serializability**](https://beginnersbook.com/2018/12/dbms-conflict-serializability/). In this article, we will discuss another type of serializability which is known as View Serializability.

## What is View Serializability?

View Serializability is a process to find out that a given [**schedule**](https://beginnersbook.com/2018/12/dbms-schedules/) is view serializable or not.

To check whether a given schedule is view serializable, we need to check whether the given schedule is View Equivalent to its serial schedule. Lets take an example to understand what I mean by that.

Given Schedule:

T1 T2

----- ------

R(X)

W(X)

R(X)

W(X)

R(Y)

W(Y)

R(Y)

W(Y)

Serial Schedule of the above given schedule:  
As we know that in [**Serial schedule**](https://beginnersbook.com/2018/12/dbms-schedules/) a transaction only starts when the current running transaction is finished. So the serial schedule of the above given schedule would look like this:

T1 T2

----- ------

R(X)

W(X)

R(Y)

W(Y)

R(X)

W(X)

R(Y)

W(Y)

If we can prove that the given schedule is View Equivalent to its serial schedule then the given schedule is called view Serializable.

## Why we need View Serializability?

We know that a serial schedule never leaves the database in inconsistent state because there are no concurrent transactions execution. However a non-serial schedule can leave the database in inconsistent state because there are multiple transactions running concurrently. By checking that a given non-serial schedule is view serializable, we make sure that it is a consistent schedule.

You may be wondering instead of checking that a non-serial schedule is serializable or not, can’t we have serial schedule all the time? The answer is no, because concurrent execution of transactions fully utilize the system resources and are considerably faster compared to serial schedules.

## View Equivalent

Lets learn how to check whether the two schedules are view equivalent.

Two schedules T1 and T2 are said to be view equivalent, if they satisfy all the following conditions:

1. Initial Read: Initial read of each data item in transactions must match in both schedules. For example, if transaction T1 reads a data item X before transaction T2 in schedule S1 then in schedule S2, T1 should read X before T2.

Read vs Initial Read: You may be confused by the term initial read. Here initial read means the first read operation on a data item, for example, a data item X can be read multiple times in a schedule but the first read operation on X is called the initial read. This will be more clear once we will get to the example in the next section of this same article.

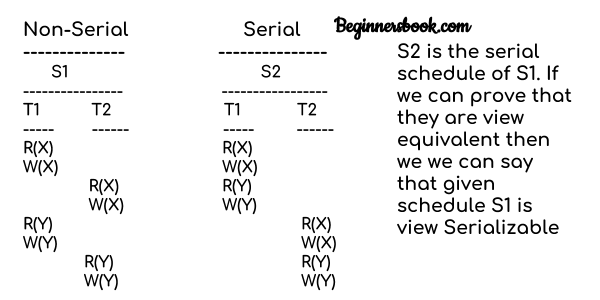
2. Final Write: Final write operations on each data item must match in both the schedules. For example, a data item X is last written by Transaction T1 in schedule S1 then in S2, the last write operation on X should be performed by the transaction T1.

3. Update Read: If in schedule S1, the transaction T1 is reading a data item updated by T2 then in schedule S2, T1 should read the value after the write operation of T2 on same data item. For example, In schedule S1, T1 performs a read operation on X after the write operation on X by T2 then in S2, T1 should read the X after T2 performs write on X.

## View Serializable

If a schedule is view equivalent to its serial schedule then the given schedule is said to be View Serializable. Lets take an example.

### View Serializable Example



Lets check the three conditions of view serializability:

#### Initial Read

In schedule S1, transaction T1 first reads the data item X. In S2 also transaction T1 first reads the data item X.

Lets check for Y. In schedule S1, transaction T1 first reads the data item Y. In S2 also the first read operation on Y is performed by T1.

We checked for both data items X & Y and the initial read condition is satisfied in S1 & S2.

#### Final Write

In schedule S1, the final write operation on X is done by transaction T2. In S2 also transaction T2 performs the final write on X.

Lets check for Y. In schedule S1, the final write operation on Y is done by transaction T2. In schedule S2, final write on Y is done by T2.

We checked for both data items X & Y and the final write condition is satisfied in S1 & S2.

#### Update Read

In S1, transaction T2 reads the value of X, written by T1. In S2, the same transaction T2 reads the X after it is written by T1.

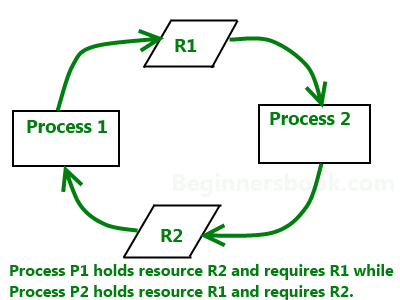
In S1, transaction T2 reads the value of Y, written by T1. In S2, the same transaction T2 reads the value of Y after it is updated by T1.

The update read condition is also satisfied for both the schedules.

Result: Since all the three conditions that checks whether the two schedules are view equivalent are satisfied in this example, which means S1 and S2 are view equivalent. Also, as we know that the schedule S2 is the serial schedule of S1, thus we can say that the schedule S1 is view serializable schedule.

# Deadlock in DBMS

BY CHAITANYA SINGH | FILED UNDER: [**DBMS**](https://beginnersbook.com/category/dbms/)

A deadlock is a condition wherein two or more tasks are waiting for each other in order to be finished but none of the task is willing to give up the resources that other task needs. In this situation no task ever gets finished and is in waiting state forever.  


## Coffman conditions

Coffman stated four conditions for a deadlock occurrence. A deadlock may occur if all the following conditions holds true.

* Mutual exclusion condition: There must be at least one resource that cannot be used by more than one process at a time.
* Hold and wait condition: A process that is holding a resource can request for additional resources that are being held by other processes in the system.
* No preemption condition: A resource cannot be forcibly taken from a process. Only the process can release a resource that is being held by it.
* Circular wait condition: A condition where one process is waiting for a resource that is being held by second process and second process is waiting for third process ….so on and the last process is waiting for the first process. Thus making a circular chain of waiting.

## Deadlock Handling

### Ignore the deadlock (Ostrich algorithm)

Did that made you laugh? You may be wondering how ignoring a deadlock can come under deadlock handling. But to let you know that the windows you are using on your PC, uses this approach of deadlock handling and that is reason sometimes it hangs up and you have to reboot it to get it working. Not only Windows but UNIX also uses this approach.

The question is why? Why instead of dealing with a deadlock they ignore it and why this is being called as Ostrich algorithm?

Well! Let me answer the second question first, This is known as Ostrich algorithm because in this approach we ignore the deadlock and pretends that it would never occur, just like Ostrich behavior “to stick one’s head in the sand and pretend there is no problem.”

Let’s discuss why we ignore it: When it is believed that deadlocks are very rare and cost of deadlock handling is higher, in that case ignoring is better solution than handling it. For example: Let’s take the operating system example – If the time requires handling the deadlock is higher than the time requires rebooting the windows then rebooting would be a preferred choice considering that deadlocks are very rare in windows.

### Deadlock detection

Resource scheduler is one that keeps the track of resources allocated to and requested by processes. Thus, if there is a deadlock it is known to the resource scheduler. This is how a deadlock is detected.

Once a deadlock is detected it is being corrected by following methods:

* Terminating processes involved in deadlock: Terminating all the processes involved in deadlock or terminating process one by one until deadlock is resolved can be the solutions but both of these approaches are not good. Terminating all processes cost high and partial work done by processes gets lost. Terminating one by one takes lot of time because each time a process is terminated, it needs to check whether the deadlock is resolved or not. Thus, the best approach is considering process age and priority while terminating them during a deadlock condition.
* Resource Preemption: Another approach can be the preemption of resources and allocation of them to the other processes until the deadlock is resolved.

### Deadlock prevention

We have learnt that if all the four Coffman conditions hold true then a deadlock occurs so preventing one or more of them could prevent the deadlock.

* Removing mutual exclusion: All resources must be sharable that means at a time more than one processes can get a hold of the resources. That approach is practically impossible.
* Removing hold and wait condition: This can be removed if the process acquires all the resources that are needed before starting out. Another way to remove this to enforce a rule of requesting resource when there are none in held by the process.
* Preemption of resources: Preemption of resources from a process can result in rollback and thus this needs to be avoided in order to maintain the consistency and stability of the system.
* Avoid circular wait condition: This can be avoided if the resources are maintained in a hierarchy and process can hold the resources in increasing order of precedence. This avoid circular wait. Another way of doing this to force one resource per process rule – A process can request for a resource once it releases the resource currently being held by it. This avoids the circular wait.

## Deadlock Avoidance

Deadlock can be avoided if resources are allocated in such a way that it avoids the deadlock occurrence. There are two algorithms for deadlock avoidance.

* Wait/Die
* Wound/Wait

Here is the table representation of resource allocation for each algorithm. Both of these algorithms take process age into consideration while determining the best possible way of resource allocation for deadlock avoidance.

|  |  |  |
| --- | --- | --- |
|  | Wait/Die | Wound/Wait |
| Older process needs a resource held by younger process | Older process waits | Younger process dies |
| Younger process needs a resource held by older process | Younger process dies | Younger process waits |

Once of the famous deadlock avoidance algorithm is Banker’s algorithm

# 

# Transaction

* The transaction is a set of logically related operation. It contains a group of tasks.
* A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Example: Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

1. Open\_Account(X)
2. Old\_Balance = X.balance
3. New\_Balance = Old\_Balance - 800
4. X.balance = New\_Balance
5. Close\_Account(X)

Y's Account

1. Open\_Account(Y)
2. Old\_Balance = Y.balance
3. New\_Balance = Old\_Balance + 800
4. Y.balance = New\_Balance
5. Close\_Account(Y)

## Operations of Transaction:

Following are the main operations of transaction:

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.

Write(X): Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. 1.  R(X);
2. 2.  X = X - 500;
3. 3.  W(X);

Let's assume the value of X before starting of the transaction is 4000.

* The first operation reads X's value from database and stores it in a buffer.
* The second operation will decrease the value of X by 500. So buffer will contain 3500.
* The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

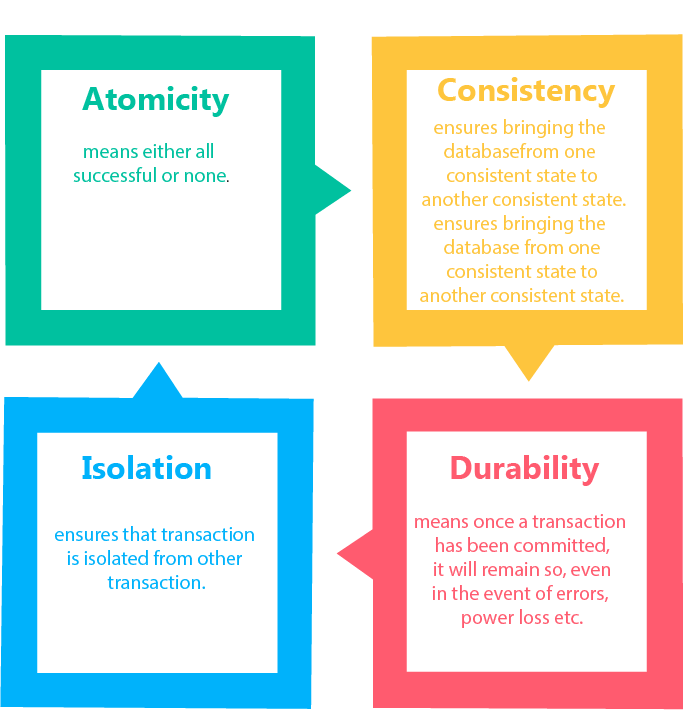
Rollback: It is used to undo the work done.

# Transaction property

The transaction has the four properties. These are used to maintain consistency in a database, before and after the transaction.

## Property of Transaction

1. Atomicity
2. Consistency
3. Isolation
4. Durability



## Atomicity

* It states that all operations of the transaction take place at once if not, the transaction is aborted.
* There is no midway, i.e., the transaction cannot occur partially. Each transaction is treated as one unit and either run to completion or is not executed at all.

Atomicity involves the following two operations:

Abort: If a transaction aborts then all the changes made are not visible.

Commit: If a transaction commits then all the changes made are visible.

Example: Let's assume that following transaction T consisting of T1 and T2. A consists of Rs 600 and B consists of Rs 300. Transfer Rs 100 from account A to account B.

|  |  |
| --- | --- |
| T1 | T2 |
| Read(A) A:= A-100 Write(A) | Read(B) Y:= Y+100 Write(B) |

After completion of the transaction, A consists of Rs 500 and B consists of Rs 400.

If the transaction T fails after the completion of transaction T1 but before completion of transaction T2, then the amount will be deducted from A but not added to B. This shows the inconsistent database state. In order to ensure correctness of database state, the transaction must be executed in entirety.

## Consistency

* The integrity constraints are maintained so that the database is consistent before and after the transaction.
* The execution of a transaction will leave a database in either its prior stable state or a new stable state.
* The consistent property of database states that every transaction sees a consistent database instance.
* The transaction is used to transform the database from one consistent state to another consistent state.

For example: The total amount must be maintained before or after the transaction.

1. Total before T occurs = 600+300=900
2. Total after T occurs= 500+400=900

Therefore, the database is consistent. In the case when T1 is completed but T2 fails, then inconsistency will occur.

## Isolation

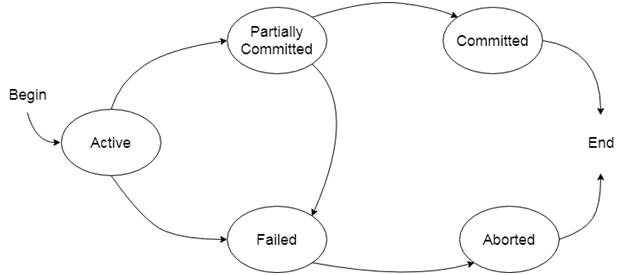
* It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
* In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
* The concurrency control subsystem of the DBMS enforced the isolation property.

## Durability

* The durability property is used to indicate the performance of the database's consistent state. It states that the transaction made the permanent changes.
* They cannot be lost by the erroneous operation of a faulty transaction or by the system failure. When a transaction is completed, then the database reaches a state known as the consistent state. That consistent state cannot be lost, even in the event of a system's failure.
* The recovery subsystem of the DBMS has the responsibility of Durability property.

States of Transaction

In a database, the transaction can be in one of the following states -



### Active state

* The active state is the first state of every transaction. In this state, the transaction is being executed.
* For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

### Partially committed

* In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
* In the total mark calculation example, a final display of the total marks step is executed in this state.

### Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

### Failed state

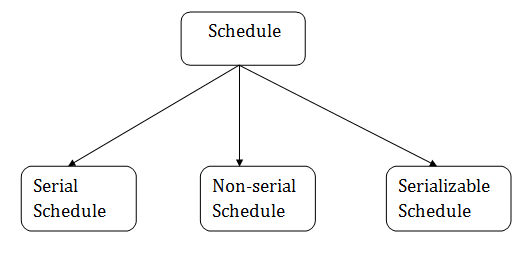
* If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
* In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

### Aborted

* If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
* If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
* After aborting the transaction, the database recovery module will select one of the two operations:
  1. Re-start the transaction
  2. Kill the transaction

# Schedule

A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.



## 1. Serial Schedule

The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

For example: Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:

1. Execute all the operations of T1 which was followed by all the operations of T2.
2. Execute all the operations of T1 which was followed by all the operations of T2.

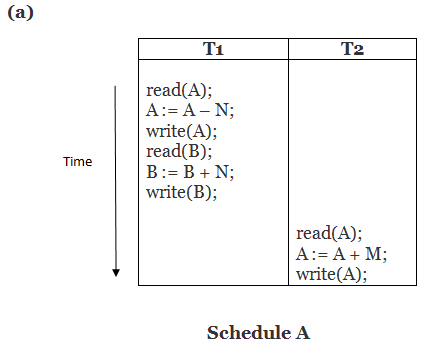
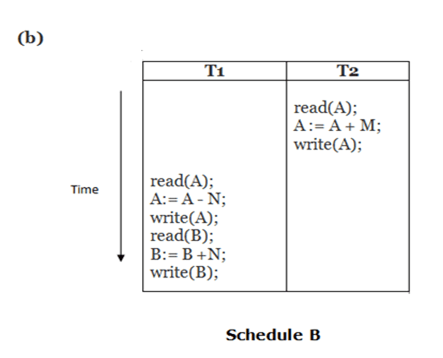
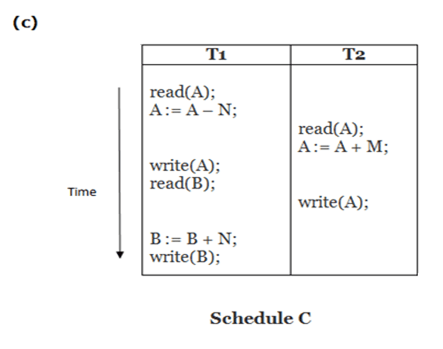
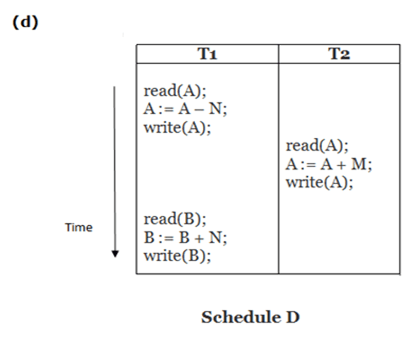
* In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
* In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

## 2. Non-serial Schedule

* If interleaving of operations is allowed, then there will be non-serial schedule.
* It contains many possible orders in which the system can execute the individual operations of the transactions.
* In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

## 3. Serializable schedule

* The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
* It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
* A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

Here,

Schedule A and Schedule B are serial schedule.

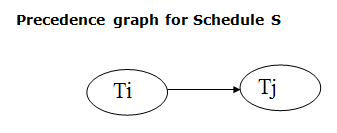
Schedule C and Schedule D are Non-serial schedule.

Testing of Serializability

Serialization Graph is used to test the Serializability of a schedule.

Assume a schedule S. For S, we construct a graph known as precedence graph. This graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule. The set of edges is used to contain all edges Ti ->Tj for which one of the three conditions holds:

1. Create a node Ti → Tj if Ti executes write (Q) before Tj executes read (Q).
2. Create a node Ti → Tj if Ti executes read (Q) before Tj executes write (Q).
3. Create a node Ti → Tj if Ti executes write (Q) before Tj executes write (Q).



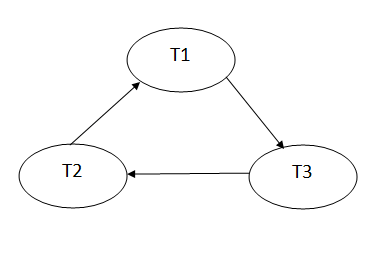
* If a precedence graph contains a single edge Ti → Tj, then all the instructions of Ti are executed before the first instruction of Tj is executed.
* If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

For example:

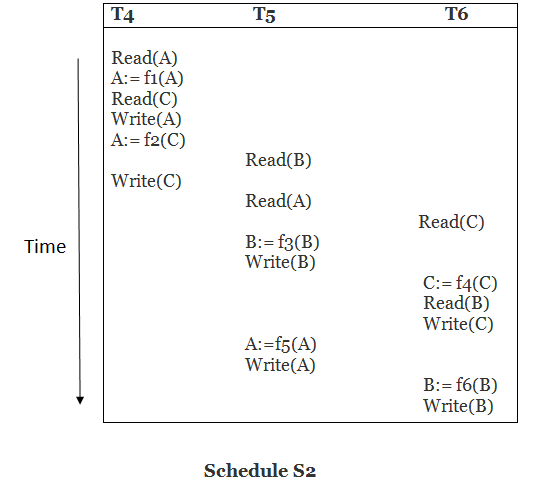
Explanation:

Read(A): In T1, no subsequent writes to A, so no new edges  
Read(B): In T2, no subsequent writes to B, so no new edges  
Read(C): In T3, no subsequent writes to C, so no new edges  
Write(B): B is subsequently read by T3, so add edge T2 → T3  
Write(C): C is subsequently read by T1, so add edge T3 → T1  
Write(A): A is subsequently read by T2, so add edge T1 → T2  
Write(A): In T2, no subsequent reads to A, so no new edges  
Write(C): In T1, no subsequent reads to C, so no new edges  
Write(B): In T3, no subsequent reads to B, so no new edges

### Precedence graph for schedule S1:



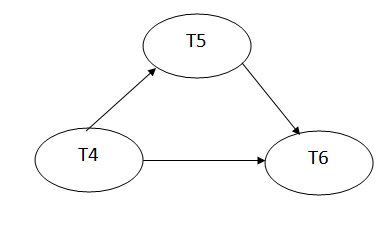
The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non-serializable.



Explanation:

Read(A): In T4,no subsequent writes to A, so no new edges  
Read(C): In T4, no subsequent writes to C, so no new edges  
Write(A): A is subsequently read by T5, so add edge T4 → T5  
Read(B): In T5,no subsequent writes to B, so no new edges  
Write(C): C is subsequently read by T6, so add edge T4 → T6  
Write(B): A is subsequently read by T6, so add edge T5 → T6  
Write(C): In T6, no subsequent reads to C, so no new edges  
Write(A): In T5, no subsequent reads to A, so no new edges  
Write(B): In T6, no subsequent reads to B, so no new edges

### Precedence graph for schedule S2:



The precedence graph for schedule S2 contains no cycle that's why ScheduleS2 is serializable.

# Conflict Serializable Schedule

* A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.
* The schedule will be a conflict serializable if it is conflict equivalent to a serial schedule.

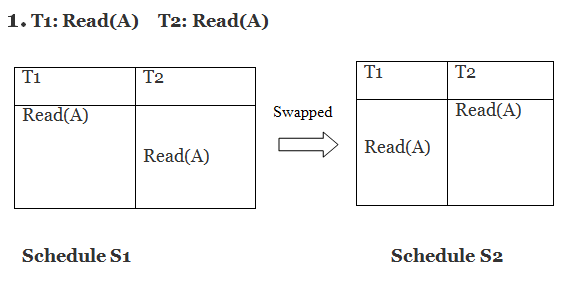
## Conflicting Operations

The two operations become conflicting if all conditions satisfy:

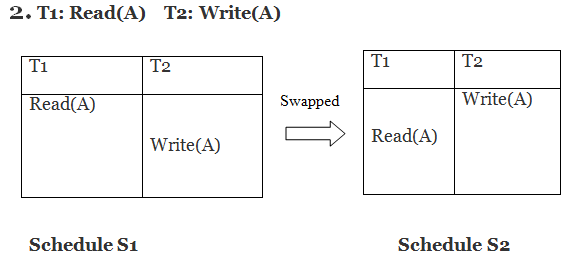
1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operation.

### Example:

Swapping is possible only if S1 and S2 are logically equal.



Here, S1 = S2. That means it is non-conflict.



Here, S1 ≠ S2. That means it is conflict.

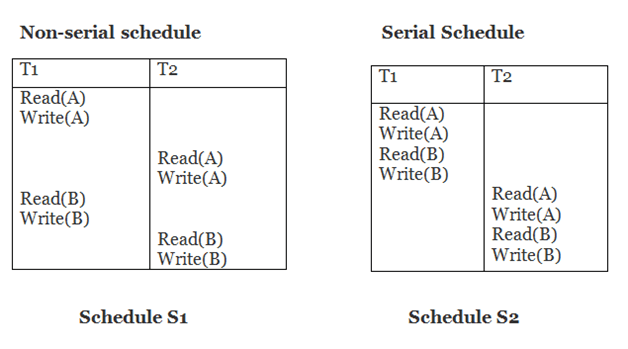
## Conflict Equivalent

In the conflict equivalent, one can be transformed to another by swapping non-conflicting operations. In the given example, S2 is conflict equivalent to S1 (S1 can be converted to S2 by swapping non-conflicting operations).

Two schedules are said to be conflict equivalent if and only if:

1. They contain the same set of the transaction.
2. If each pair of conflict operations are ordered in the same way.

### Example:



Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2. Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.

After swapping of non-conflict operations, the schedule S1 becomes:

|  |  |
| --- | --- |
| T1 | T2 |
| Read(A) Write(A) Read(B) Write(B) | Read(A) Write(A) Read(B) Write(B) |

Since, S1 is conflict serializable.

View Serializability

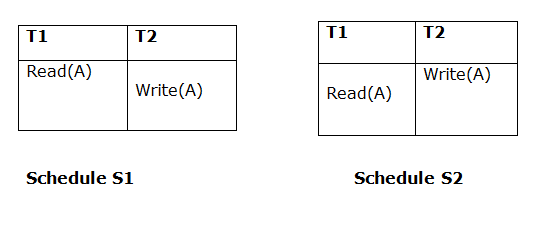
* A schedule will view serializable if it is view equivalent to a serial schedule.
* If a schedule is conflict serializable, then it will be view serializable.
* The view serializable which does not conflict serializable contains blind writes.

## View Equivalent

Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

### 1. Initial Read

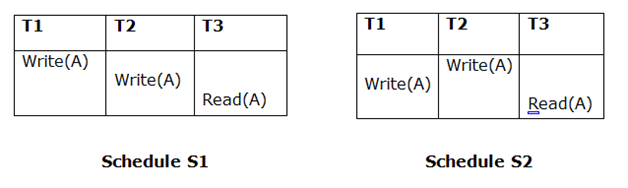
An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.



Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

### 2. Updated Read

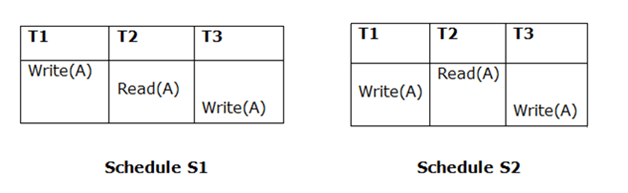
In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.



Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

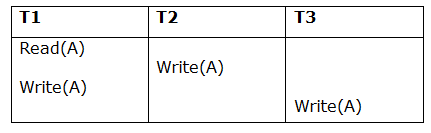
### 3. Final Write

A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.



Above two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

Example:

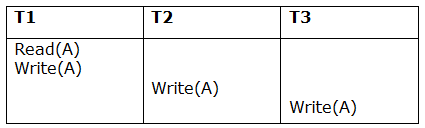


Schedule S

With 3 transactions, the total number of possible schedule

1. = 3! = 6
2. S1 = <T1 T2 T3>
3. S2 = <T1 T3 T2>
4. S3 = <T2 T3 T1>
5. S4 = <T2 T1 T3>
6. S5 = <T3 T1 T2>
7. S6 = <T3 T2 T1>

Taking first schedule S1:



Schedule S1

Step 1: final updation on data items

In both schedules S and S1, there is no read except the initial read that's why we don't need to check that condition.

Step 2: Initial Read

The initial read operation in S is done by T1 and in S1, it is also done by T1.

Step 3: Final Write

The final write operation in S is done by T3 and in S1, it is also done by T3. So, S and S1 are view Equivalent.

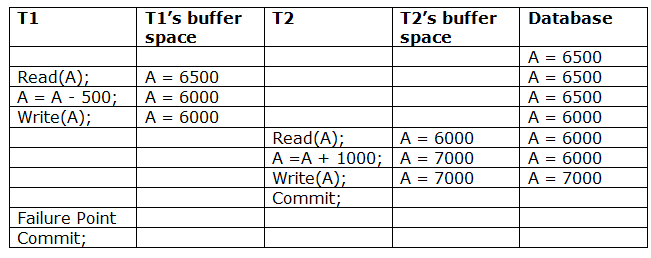
The first schedule S1 satisfies all three conditions, so we don't need to check another schedule.

Hence, view equivalent serial schedule is:

1. T1    →      T2    →    T3

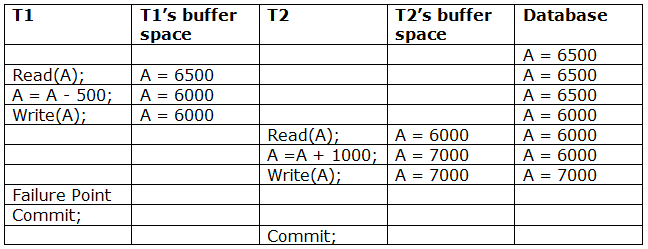
# Recoverability of Schedule

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.



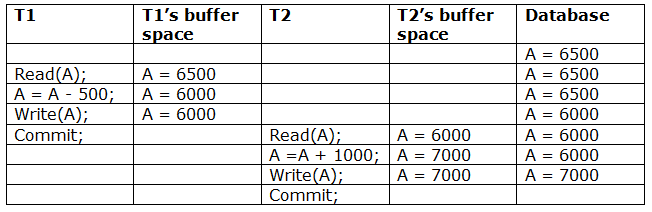
The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.



The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.



The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

# Failure Classification

To find that where the problem has occurred, we generalize a failure into the following categories:

1. Transaction failure
2. System crash
3. Disk failure

### 1. Transaction failure

The transaction failure occurs when it fails to execute or when it reaches a point from where it can't go any further. If a few transaction or process is hurt, then this is called as transaction failure.

Reasons for a transaction failure could be -

* 1. Logical errors: If a transaction cannot complete due to some code error or an internal error condition, then the logical error occurs.
  2. Syntax error: It occurs where the DBMS itself terminates an active transaction because the database system is not able to execute it. For example, The system aborts an active transaction, in case of deadlock or resource unavailability.

### 2. System Crash

* 1. System failure can occur due to power failure or other hardware or software failure. Example: Operating system error.

Fail-stop assumption: In the system crash, non-volatile storage is assumed not to be corrupted.

### 3. Disk Failure

* 1. It occurs where hard-disk drives or storage drives used to fail frequently. It was a common problem in the early days of technology evolution.
  2. Disk failure occurs due to the formation of bad sectors, disk head crash, and unreachability to the disk or any other failure, which destroy all or part of disk storage.

# Log-Based Recovery

* The log is a sequence of records. Log of each transaction is maintained in some stable storage so that if any failure occurs, then it can be recovered from there.
* If any operation is performed on the database, then it will be recorded in the log.
* But the process of storing the logs should be done before the actual transaction is applied in the database.

Let's assume there is a transaction to modify the City of a student. The following logs are written for this transaction.

* When the transaction is initiated, then it writes 'start' log.
  1. <Tn, Start>
* When the transaction modifies the City from 'Noida' to 'Bangalore', then another log is written to the file.
  1. <Tn, City, 'Noida', 'Bangalore' >
* When the transaction is finished, then it writes another log to indicate the end of the transaction.
  1. <Tn, Commit>

There are two approaches to modify the database:

### 1. Deferred database modification:

* The deferred modification technique occurs if the transaction does not modify the database until it has committed.
* In this method, all the logs are created and stored in the stable storage, and the database is updated when a transaction commits.

### 2. Immediate database modification:

* The Immediate modification technique occurs if database modification occurs while the transaction is still active.
* In this technique, the database is modified immediately after every operation. It follows an actual database modification.

## Recovery using Log records

When the system is crashed, then the system consults the log to find which transactions need to be undone and which need to be redone.

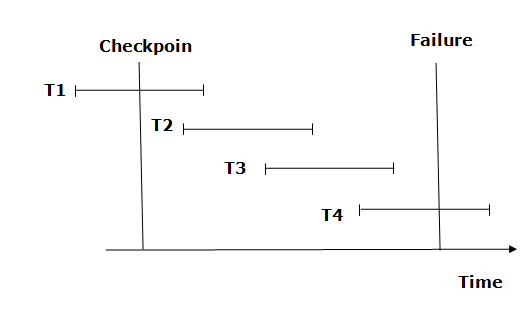
1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<Tn, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

Checkpoint

* The checkpoint is a type of mechanism where all the previous logs are removed from the system and permanently stored in the storage disk.
* The checkpoint is like a bookmark. While the execution of the transaction, such checkpoints are marked, and the transaction is executed then using the steps of the transaction, the log files will be created.
* When it reaches to the checkpoint, then the transaction will be updated into the database, and till that point, the entire log file will be removed from the file. Then the log file is updated with the new step of transaction till next checkpoint and so on.
* The checkpoint is used to declare a point before which the DBMS was in the consistent state, and all transactions were committed.

## Recovery using Checkpoint

In the following manner, a recovery system recovers the database from this failure:



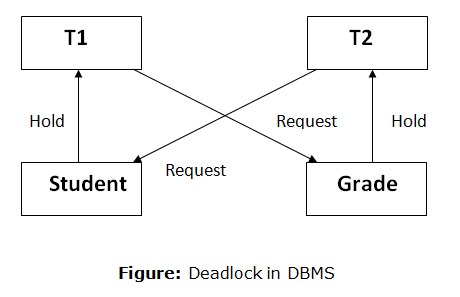
* The recovery system reads log files from the end to start. It reads log files from T4 to T1.
* Recovery system maintains two lists, a redo-list, and an undo-list.
* The transaction is put into redo state if the recovery system sees a log with <Tn, Start> and <Tn, Commit> or just <Tn, Commit>. In the redo-list and their previous list, all the transactions are removed and then redone before saving their logs.
* For example: In the log file, transaction T2 and T3 will have <Tn, Start> and <Tn, Commit>. The T1 transaction will have only <Tn, commit> in the log file. That's why the transaction is committed after the checkpoint is crossed. Hence it puts T1, T2 and T3 transaction into redo list.
* The transaction is put into undo state if the recovery system sees a log with <Tn, Start> but no commit or abort log found. In the undo-list, all the transactions are undone, and their logs are removed.
* For example: Transaction T4 will have <Tn, Start>. So T4 will be put into undo list since this transaction is not yet complete and failed amid.

Deadlock in DBMS

A deadlock is a condition where two or more transactions are waiting indefinitely for one another to give up locks. Deadlock is said to be one of the most feared complications in DBMS as no task ever gets finished and is in waiting state forever.

For example: In the student table, transaction T1 holds a lock on some rows and needs to update some rows in the grade table. Simultaneously, transaction T2 holds locks on some rows in the grade table and needs to update the rows in the Student table held by Transaction T1.

Now, the main problem arises. Now Transaction T1 is waiting for T2 to release its lock and similarly, transaction T2 is waiting for T1 to release its lock. All activities come to a halt state and remain at a standstill. It will remain in a standstill until the DBMS detects the deadlock and aborts one of the transactions.



## Deadlock Avoidance

* When a database is stuck in a deadlock state, then it is better to avoid the database rather than aborting or restating the database. This is a waste of time and resource.
* Deadlock avoidance mechanism is used to detect any deadlock situation in advance. A method like "wait for graph" is used for detecting the deadlock situation but this method is suitable only for the smaller database. For the larger database, deadlock prevention method can be used.

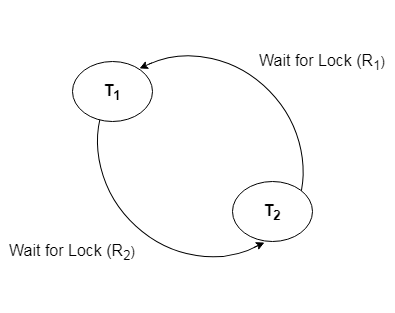
## Deadlock Detection

In a database, when a transaction waits indefinitely to obtain a lock, then the DBMS should detect whether the transaction is involved in a deadlock or not. The lock manager maintains a Wait for the graph to detect the deadlock cycle in the database.

### Wait for Graph

* This is the suitable method for deadlock detection. In this method, a graph is created based on the transaction and their lock. If the created graph has a cycle or closed loop, then there is a deadlock.
* The wait for the graph is maintained by the system for every transaction which is waiting for some data held by the others. The system keeps checking the graph if there is any cycle in the graph.

The wait for a graph for the above scenario is shown below:



## Deadlock Prevention

* Deadlock prevention method is suitable for a large database. If the resources are allocated in such a way that deadlock never occurs, then the deadlock can be prevented.
* The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

### Wait-Die scheme

In this scheme, if a transaction requests for a resource which is already held with a conflicting lock by another transaction then the DBMS simply checks the timestamp of both transactions. It allows the older transaction to wait until the resource is available for execution.

Let's assume there are two transactions Ti and Tj and let TS(T) is a timestamp of any transaction T. If T2 holds a lock by some other transaction and T1 is requesting for resources held by T2 then the following actions are performed by DBMS:

1. Check if TS(Ti) < TS(Tj) - If Ti is the older transaction and Tj has held some resource, then Ti is allowed to wait until the data-item is available for execution. That means if the older transaction is waiting for a resource which is locked by the younger transaction, then the older transaction is allowed to wait for resource until it is available.
2. Check if TS(Ti) < TS(Tj) - If Ti is older transaction and has held some resource and if Tj is waiting for it, then Tj is killed and restarted later with the random delay but with the same timestamp.

### Wound wait scheme

* In wound wait scheme, if the older transaction requests for a resource which is held by the younger transaction, then older transaction forces younger one to kill the transaction and release the resource. After the minute delay, the younger transaction is restarted but with the same timestamp.
* If the older transaction has held a resource which is requested by the Younger transaction, then the younger transaction is asked to wait until older releases it.