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.

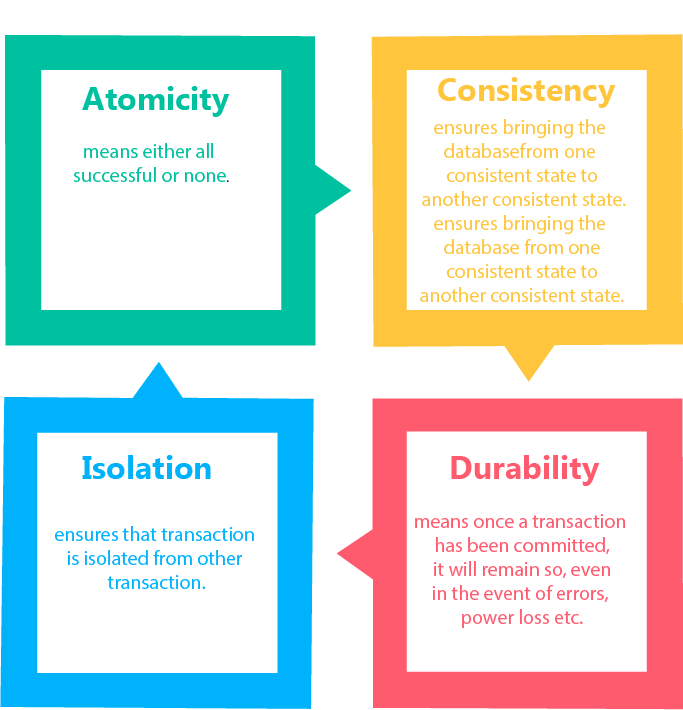
**Rollback:** It is used to undo the work done.

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

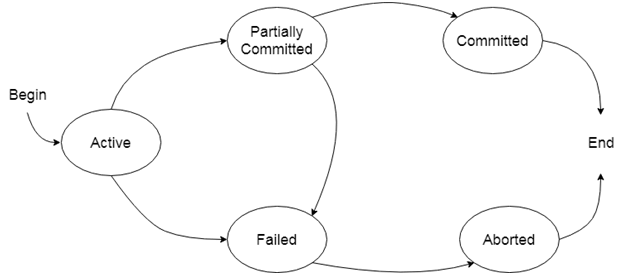
* It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
* In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction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

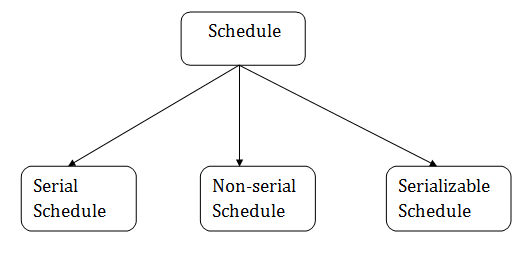
* If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
* In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

### **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.

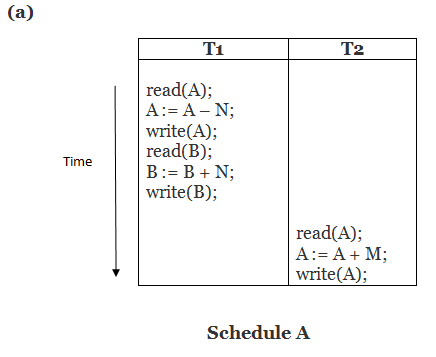
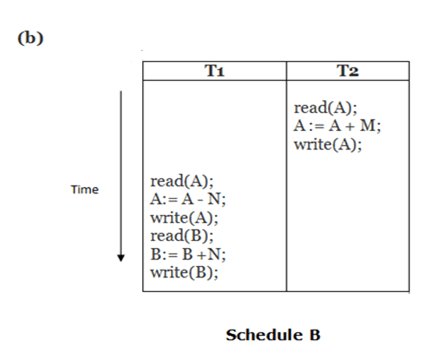
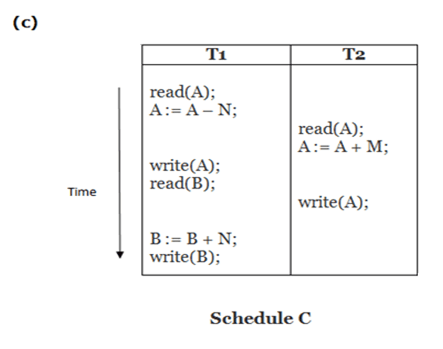
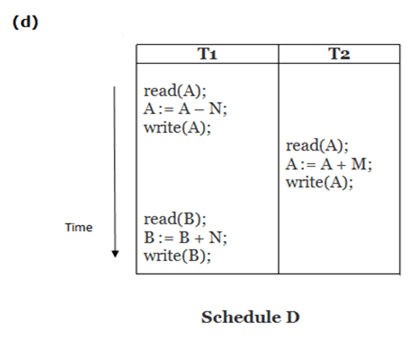
**For example:** Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:

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.

**Here,**

Schedule A and Schedule B are serial schedule.

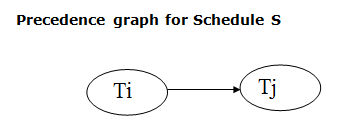
Schedule C and Schedule D are Non-serial schedule.

# Testing of Serializability

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

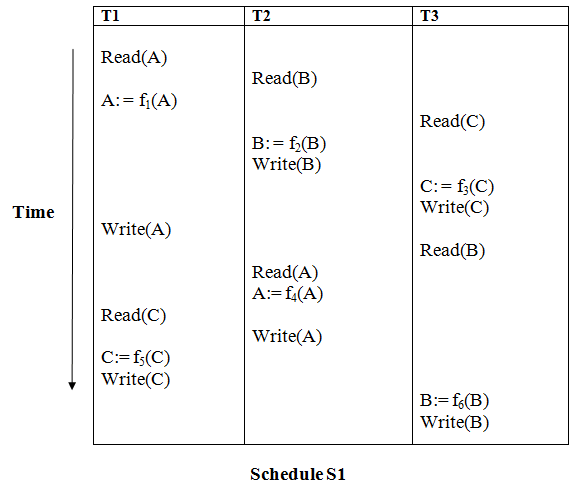
Assume a schedule S. For S, we construct a graph known as precedence graph. This graph has a pair G = (V, E), where V consists a set of vertices, and E consists a set of edges. The set of vertices is used to contain all the transactions participating in the schedule. The set of edges is used to contain 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.

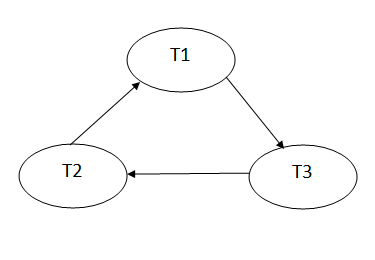
**For example:**



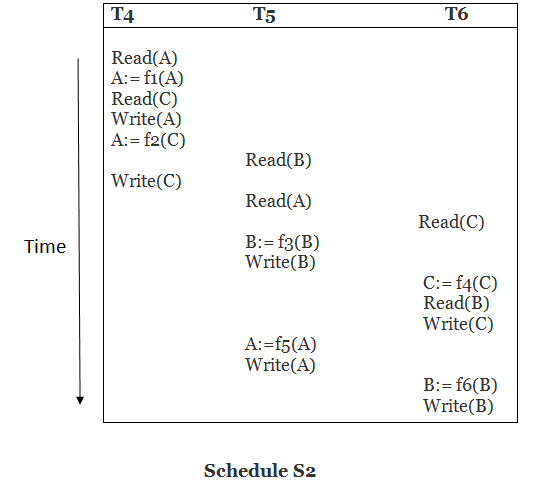
**Explanation:**

**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:**



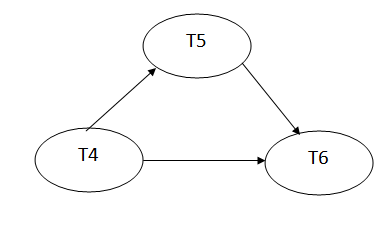
The precedence graph for schedule S1 contains a cycle that's why Schedule S1 is non-serializable.



**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.

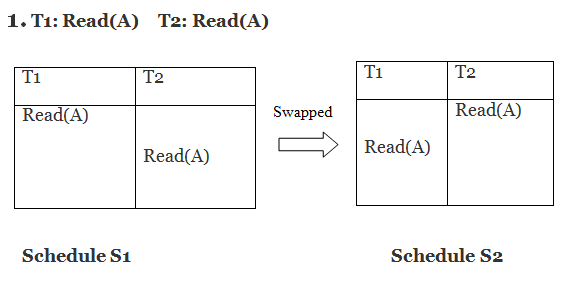
## Conflicting Operations

The two operations become conflicting if all conditions satisfy:

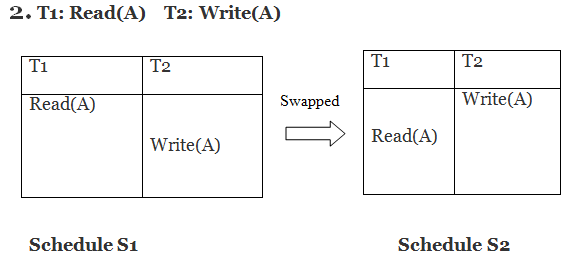
1. Both belong to separate transactions.
2. They have the same data item.
3. They contain at least one write operation.

### **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.

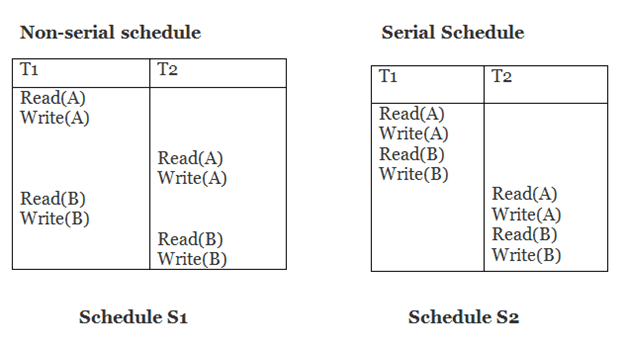
## Conflict Equivalent

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.

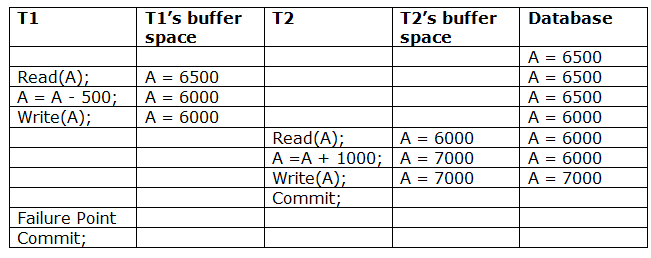
**After swapping of non-conflict operations, the schedule S1 becomes:**

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

Since, S1 is conflict serializable.

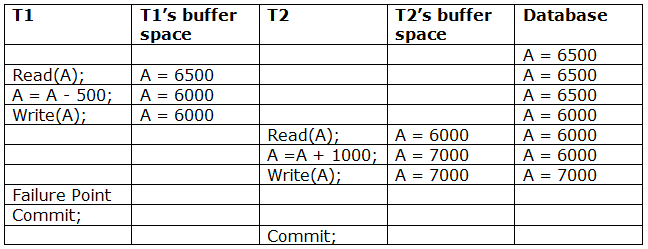
Recoverability of Schedule

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.



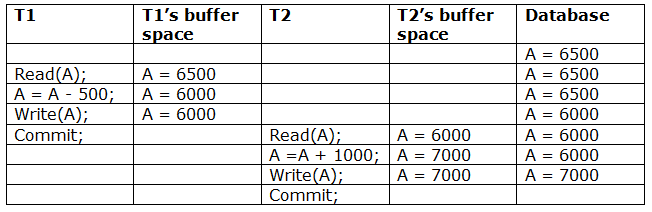
The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't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

## 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.

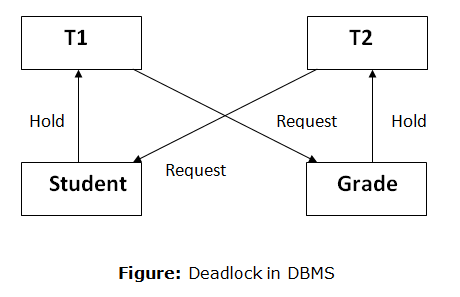
1. If the log contains the record <Ti, Start> and <Ti, Commit> or <Ti, Commit>, then the Transaction Ti needs to be redone.
2. If log contains record<Tn, Start> but does not contain the record either <Ti, commit> or <Ti, abort>, then the Transaction Ti needs to be undone.

# 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.

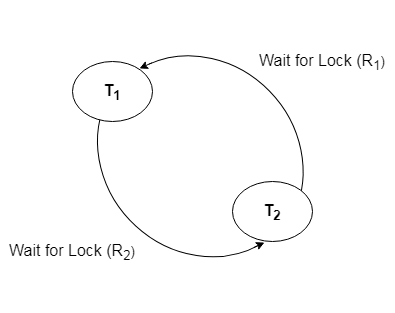
## Deadlock Detection

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.