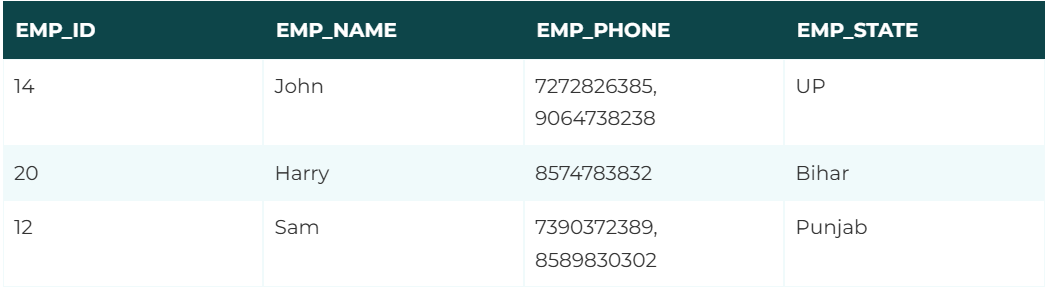
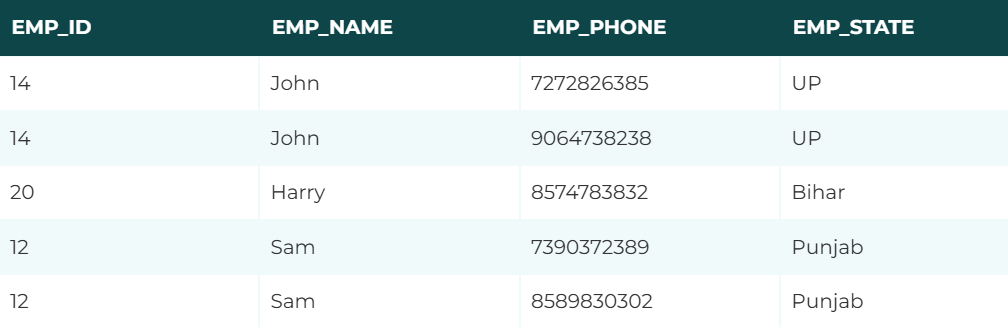
* **Normalization**
* Normalization is the process of organizing the data in the database.
* Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
* Normalization divides the larger table into smaller and links them using relationships.
* The normal form is used to reduce redundancy from the database table.
* **Why do we need Normalization:**
* The main reason for normalizing the relations is removing these anomalies.
* Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure.
* **Data modification anomalies can be categorized into three types:**

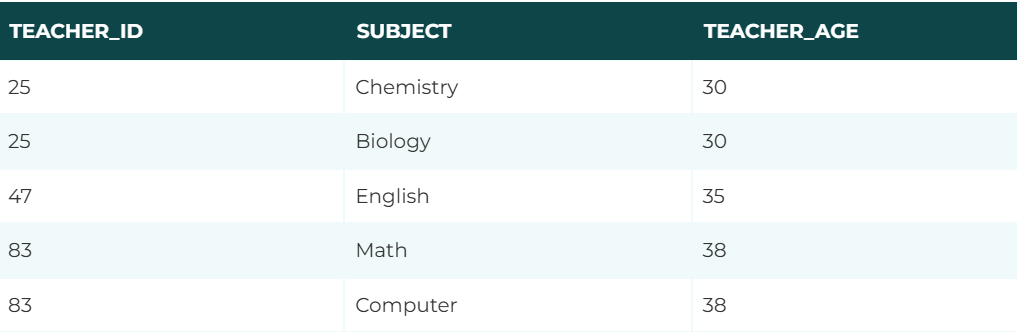
1. Insertion Anomaly: Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
2. Deletion Anomaly: The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
3. Updatation Anomaly: The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

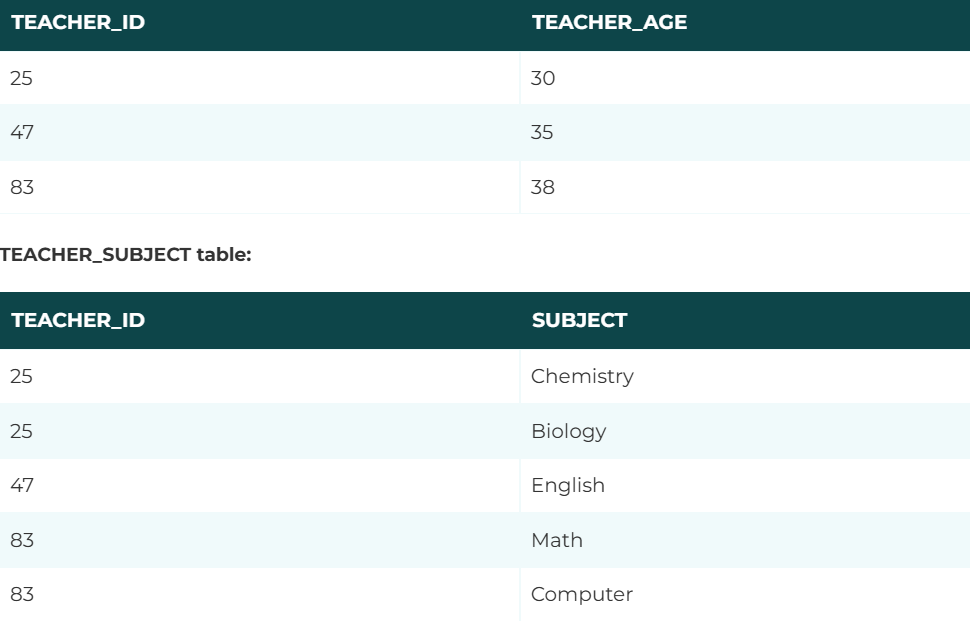
* **Types of Normal Forms:**
* Normalization works through a series of stages called Normal forms. The normal forms apply to individual relations. The relation is said to be in particular normal form if it satisfies constraints.
* Following are the various types of Normal forms:
* **1NF**
* A relation will be 1NF if it contains an atomic value.
* It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
* First normal form disallows the multi-valued attribute, composite attribute, and their combinations.
* **Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP\_PHONE.



The decomposition of the EMPLOYEE table into 1NF has been shown below:



* **2NF**
* In the 2NF, relational must be in 1NF.
* In the second normal form, all non-key attributes are fully functional dependent on the primary key
* **Example:** Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.
* 
* In the given table, non-prime attribute TEACHER\_AGE is dependent on TEACHER\_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.
* To convert the given table into 2NF, we decompose it into two tables:

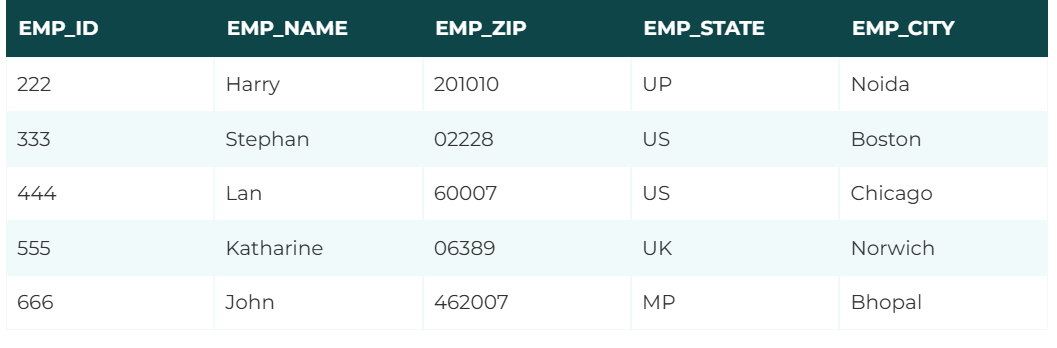


* **3NF**
* A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
* 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
* If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.
* A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

X is a super key.

Y is a prime attribute, i.e., each element of Y is part of some candidate key.

* **Example:**

****

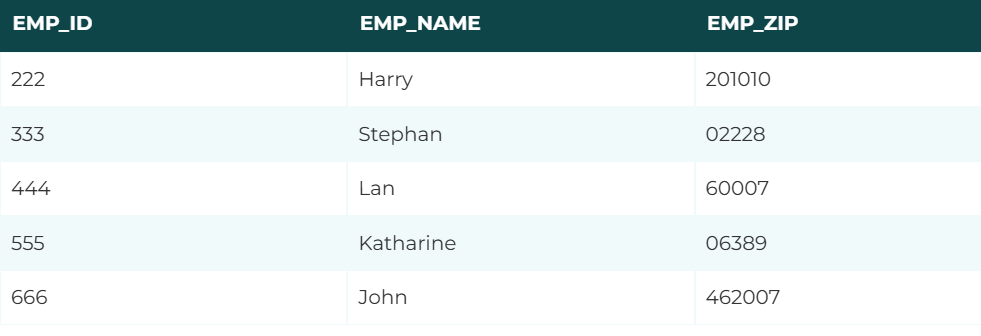
* **Super key in the table above:**

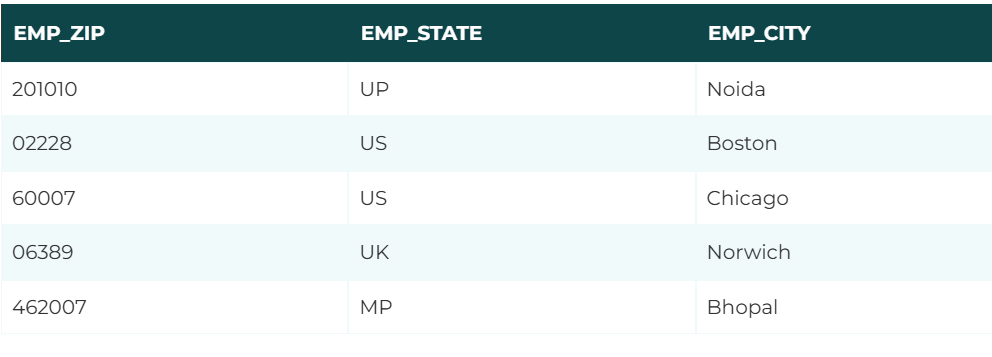
{EMP\_ID}, {EMP\_ID, EMP\_NAME}, {EMP\_ID, EMP\_NAME, EMP\_ZIP}....so on

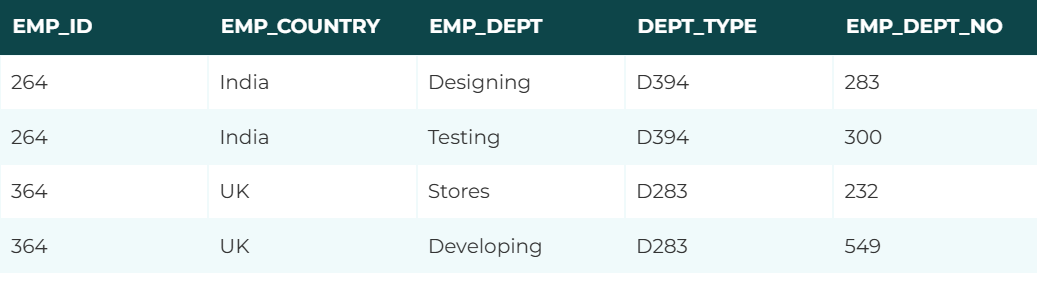
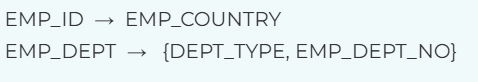
* **Candidate key:**

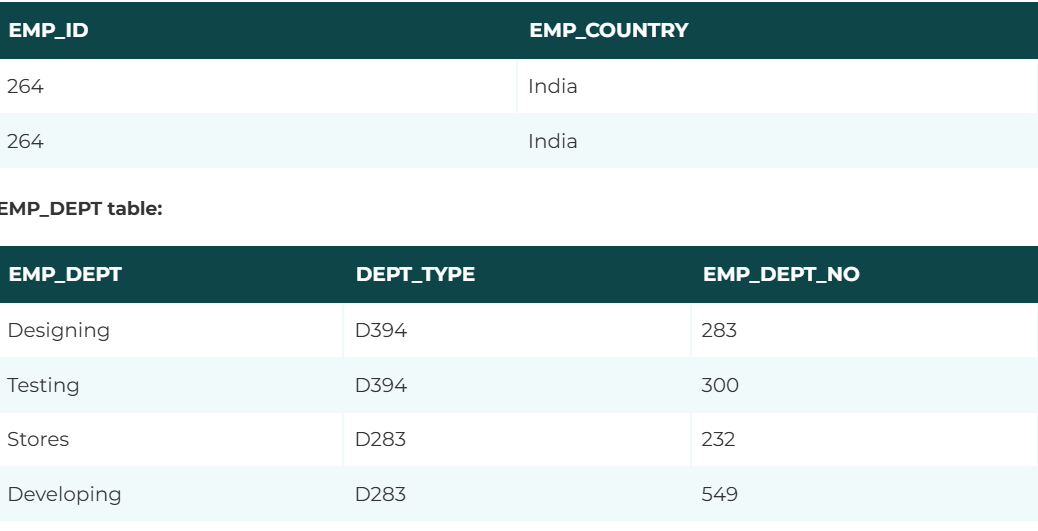
{EMP\_ID}

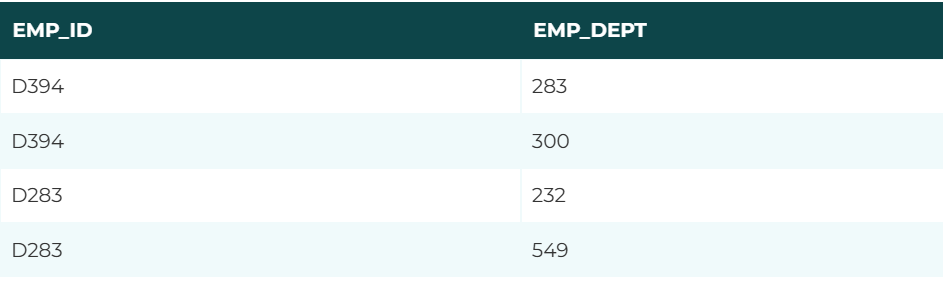
* **Non-prime attributes:** In the given table, all attributes except EMP\_ID are non-prime.
* Here, EMP\_STATE & EMP\_CITY dependent on EMP\_ZIP and EMP\_ZIP dependent on EMP\_ID. The non-prime attributes (EMP\_STATE, EMP\_CITY) transitively dependent on super key(EMP\_ID). It violates the rule of third normal form.The reduction of 2NF relation into 3NF consists of splitting the 2NF into appropriate relations such that every non-key attribute are functionally dependent on the primary key not transitively or indirectly of the respective relations.
* That's why we need to move the EMP\_CITY and EMP\_STATE to the new <EMPLOYEE\_ZIP> table, with EMP\_ZIP as a Primary key.

****

****

* **BCNF**
* BCNF is the advance version of 3NF. It is stricter than 3NF.
* A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
* For BCNF, the table should be in 3NF, and for every FD, LHS is super key.
* **Example:**
* 
* 
* **Candidate key: {EMP-ID, EMP-DEPT}**
* The table is not in BCNF because neither EMP\_DEPT nor EMP\_ID alone are keys.
* To convert the given table into BCNF, we decompose it into three tables:



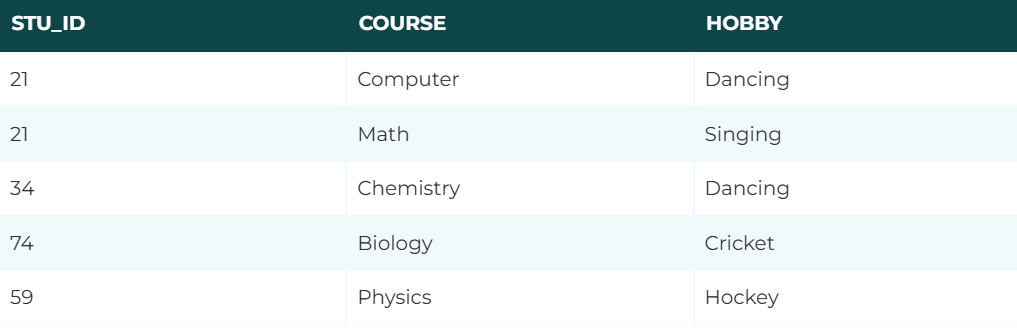


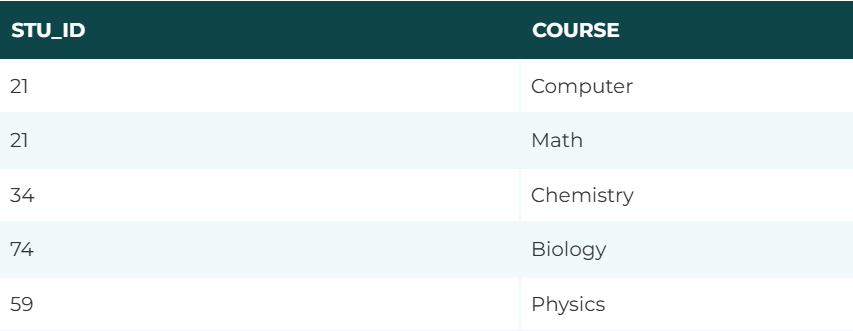
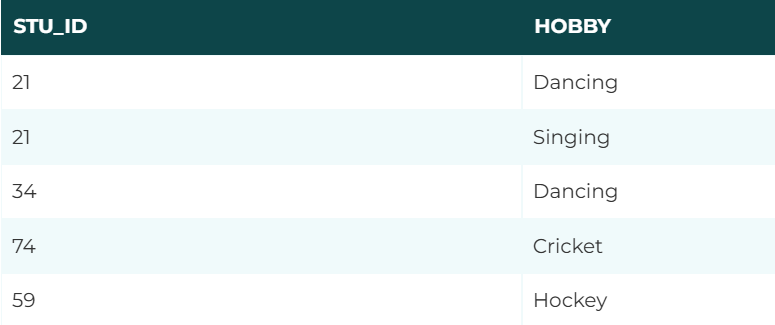
* **Functional dependencies:**

EMP\_ID   →    EMP\_COUNTRY

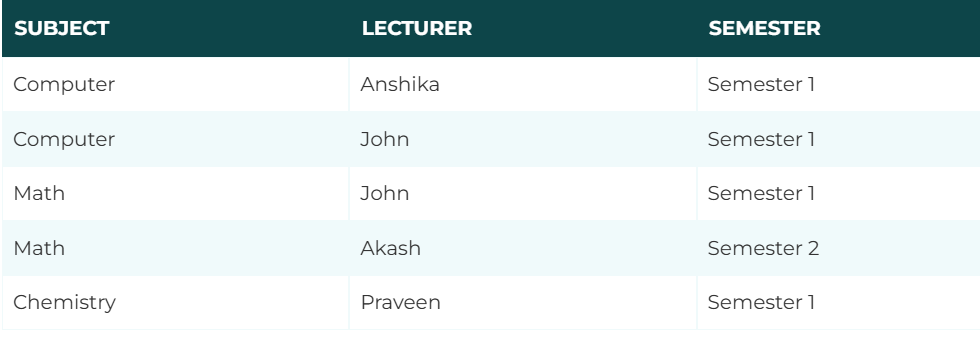
EMP\_DEPT   →   {DEPT\_TYPE, EMP\_DEPT\_NO}

* **Candidate keys:**
* **For the first table:** EMP\_ID
* **For the second table:** EMP\_DEPT  
  **For the third table:** {EMP\_ID, EMP\_DEPT}
* Now, this is in BCNF because left side part of both the functional dependencies is a key.
* **4NF**
* A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
* For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.
* Example

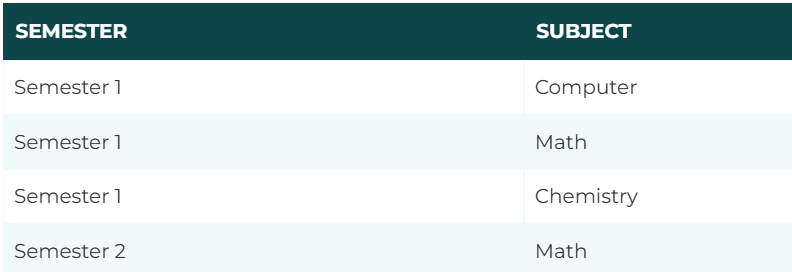


* The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.
* In the STUDENT relation, a student with STU\_ID, 21 contains two courses, Computer and Math and two hobbies, Dancing and Singing. So there is a Multi-valued dependency on STU\_ID, which leads to unnecessary repetition of data.
* So to make the above table into 4NF, we can decompose it into two tables:
* 
* 
* **5NF**
* A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
* 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
* 5NF is also known as Project-join normal form (PJ/NF).

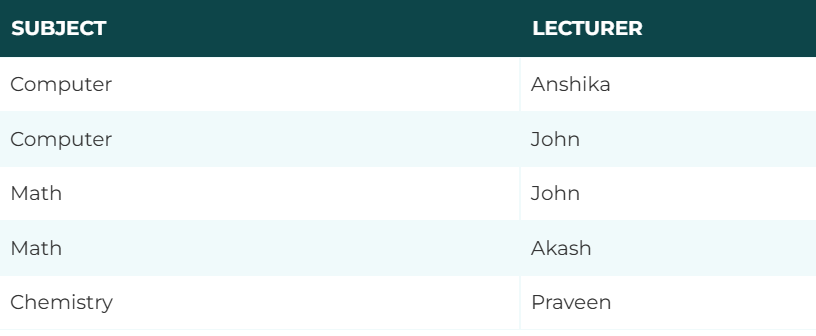
### **Example**



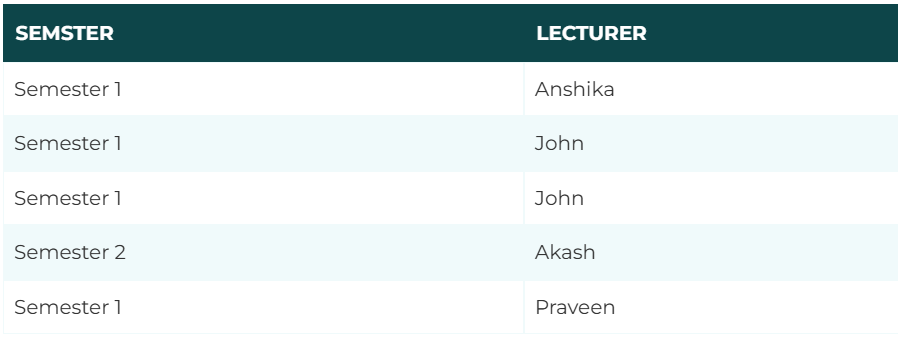
* In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.
* Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.
* So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:
* P1



P2



P3



* **Advantages of Normalization**

1. Normalization helps to minimize data redundancy.
2. Greater overall database organization.
3. Data consistency within the database.
4. Much more flexible database design.
5. Enforces the concept of relational integrity.

* **Disadvantages of Normalization**

1. You cannot start building the database before knowing what the user needs.
2. The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
3. It is very time-consuming and difficult to normalize relations of a higher degree.
4. Careless decomposition may lead to a bad database design, leading to serious problems.

* **Transaction system**
* 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

Open\_Account(X)

Old\_Balance = X.balance

New\_Balance = Old\_Balance - 800

X.balance = New\_Balance

Close\_Account(X)

Y's Account

Open\_Account(Y)

Old\_Balance = Y.balance

New\_Balance = Old\_Balance + 800

Y.balance = New\_Balance

Close\_Account(Y)

* **Operations of Transaction:**

1. **Read(X):**

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

1. **Write(X):**

* Write operation is used to write the value back to the database from the buffer.

🡺**example:**

R(X);

X = X - 500;

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.

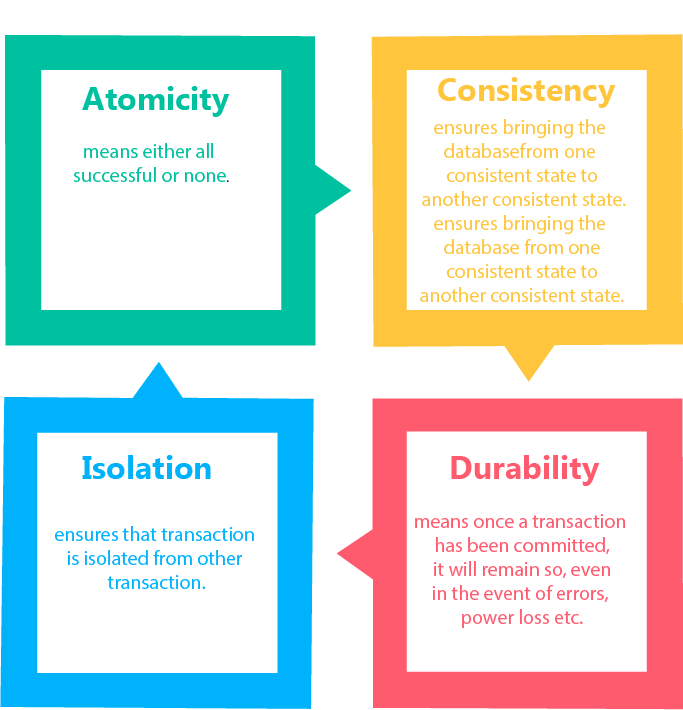
To solve this problem, we have two important operations:

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

Rollback: It is used to undo the work done.

* **Property of Transaction**

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



1. **Atomicity**

* The phrase “all or nothing” describes the first ACID property i.e. 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.
* Features of Atomicity Property:

1. Either all operations that are part of the transaction are completed or none.
2. It is maintained in the presence of disk, CPU, database software and application software failures.
3. It is maintained in the presence of deadlocks.
4. It can be locked at the system level as well as at the session level
5. Ensuring atomicity is the responsibility of the database system itself.

* 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.



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.

The mechanism for maintaining atomicity is done by the DBMS keeping track of the old values of the data on which the write operation is performed and if the transaction does not complete its execution, the old values are restored to appear as if the transaction had never been executed.

1. **Consistency**

* The phrase “no violation of integrity constraints” describes the property of 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.

Total before T occurs = 600+300=900

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.

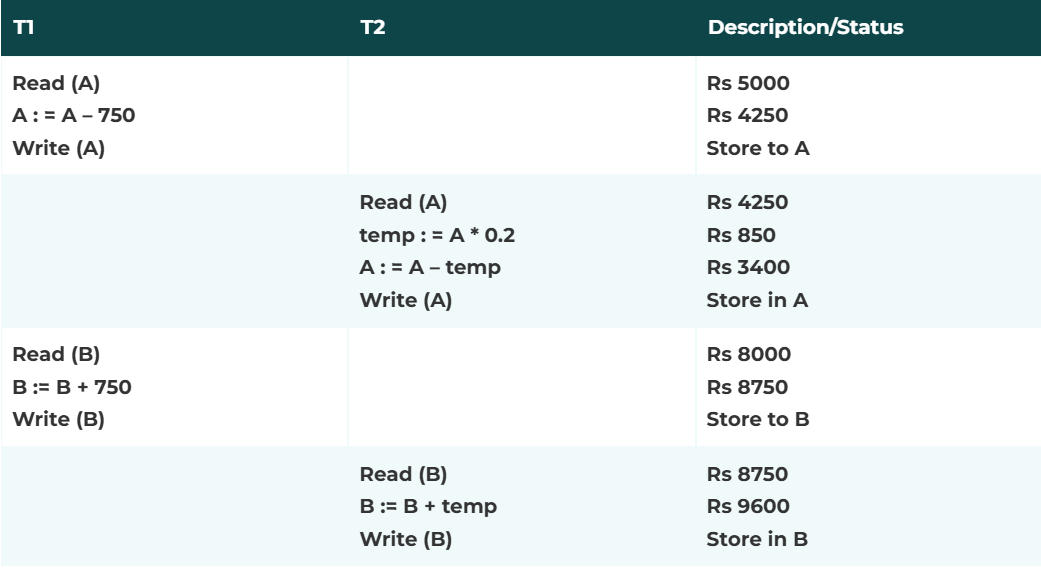
Without a consistent property, money can be credited or debited by the transaction. It is the responsibility of the application programmers who code the transaction to maintain the consistency for individual transactions by enforcing consistency constraints on the database.

1. **Isolation**

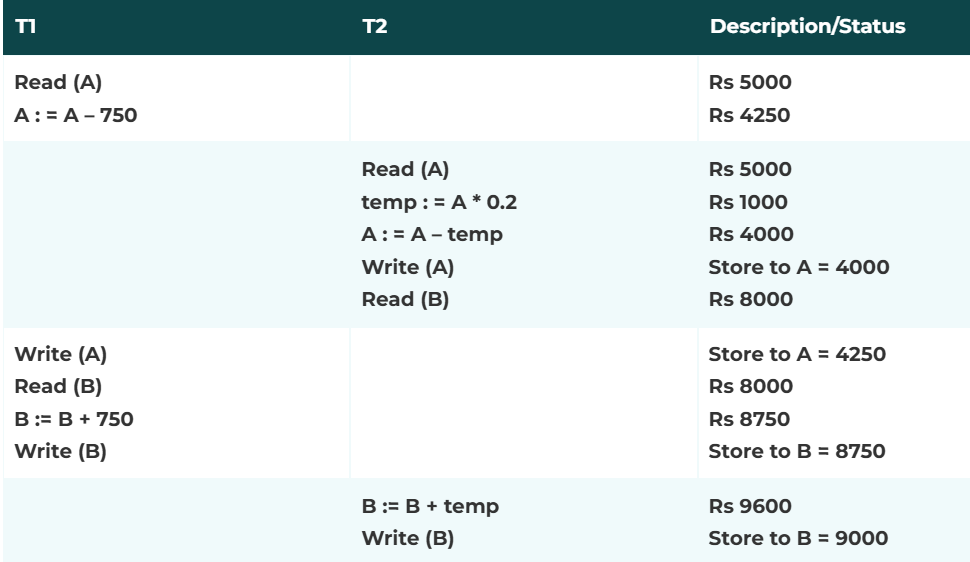
* It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
* In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
* The concurrency control subsystem of the DBMS enforced the isolation property.
* Example: Now let us consider the two transactions T1 and T2 executing concurrently. The transaction T1 is transferring amount Rs 750 from account A to account B and transaction T2 transfers 20% of amount from account A to account B. Suppose initially account balances in Account A and B are Rs 5000 and 8000 respectively.

Now the schedule for the transactions T1 and T2 running concurrently as explained in the figure below:

Schedule 1: Consistent State

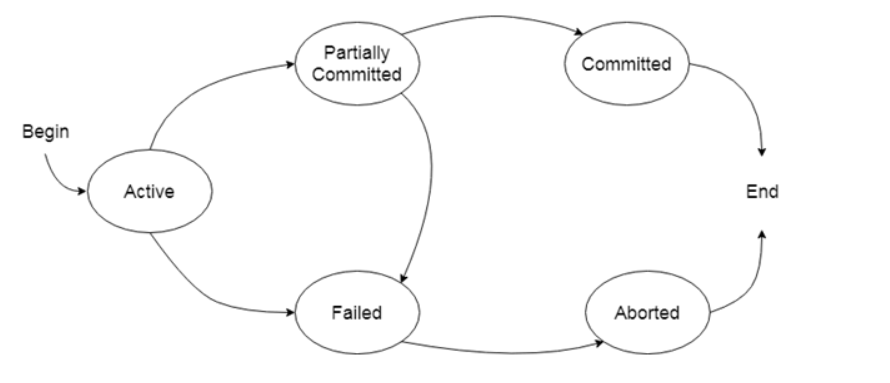


Schedule 2: Inconsistent State



* The schedule 1 results in a consistent state of the database as it shows the sum of the accounts A and B to be Rs 13000 before and after the execution of transaction.
* Now consider the schedule 2, in which the transaction T1 has performed write operation on the account A, thus updating its value to Rs 4250 by reducing the amount of Rs 750 from account A and during this time, another transaction T2 reads the balance of account A i.e. Rs 5000.
* Now it calculates the 20% amount of account A and stores it in the temp variable. Thereafter, it updates the account A balance to Rs 4000 by deducting temp variable value i.e. Rs 1000 from initial balance of account A i.e. Rs 5000. Now it also reads the balance of account B. Before executing further transaction T1 gains control and performs the write operation on account A. Now transaction T1 and the value of balance in account B so it will update the value of B to Rs 9000 instead of Rs 9750. Thus, the sum of account A and B turns out to be Rs 13250 = (4250 + 9000) which leads to inconsistency as it should be 13000.
* One of the possible solutions to the above problem is to execute the above transactions serially i.e. one after the other. But it can also lead to problems. Suppose a long transaction is being done first then all the other transactions will have to wait in the queue even through some transactions do not interfere with each other. So there is no need to wait in the queue for such transactions.
* However, concurrent execution of a group of concurrent transactions offers performance benefits. So DBMS has developed other solutions that allow the execution of multiple transactions concurrently without any problem. One of the possible methods is explained in schedule 1. This is the requirement of the isolation property which demands that data used during execution of one transaction cannot be used by the second until the first transaction is completed. Isolation property is handled by the concurrent control components of the DBMS.

1. **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.
* The DBMS maintains a record called a log that tracks all writes to the database. The log helps ensure durability. If the system crashes before the changes made by a completed transaction are written to the disk, the log is used to remember these changes when the system restarts.
* **State of transactions:**
* 

### **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**

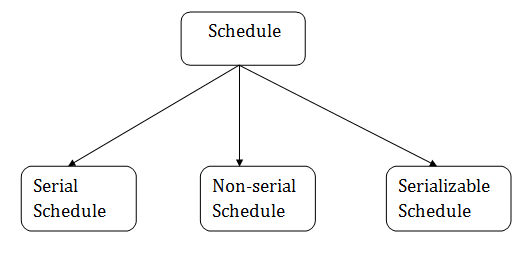
* If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
* In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

### **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
* **Schedules and serializablity**

**(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.



## 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:

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

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.

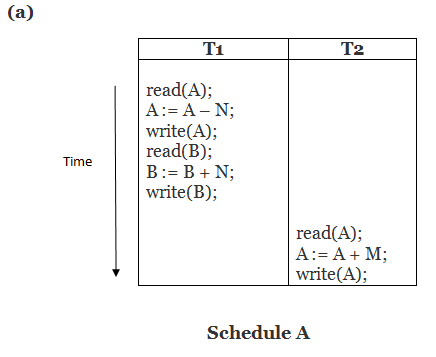
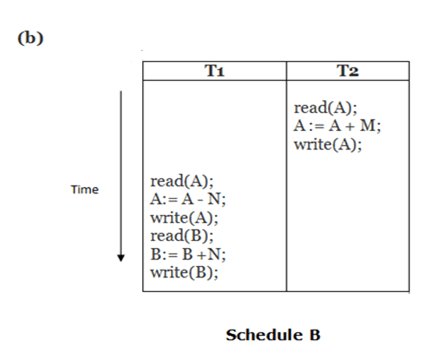
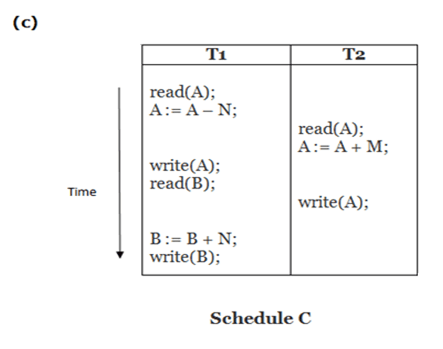
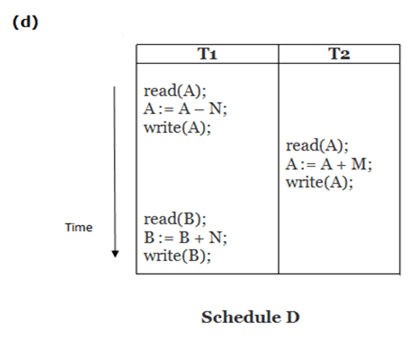
In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

## 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.

## 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,**

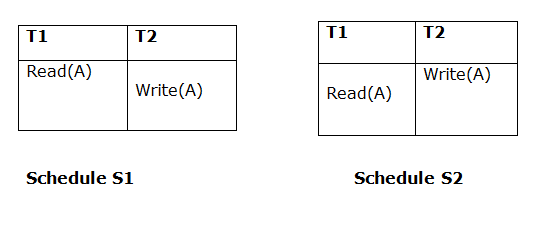
Schedule A and Schedule B are serial schedule.

Schedule C and Schedule D are Non-serial schedule.

* **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**

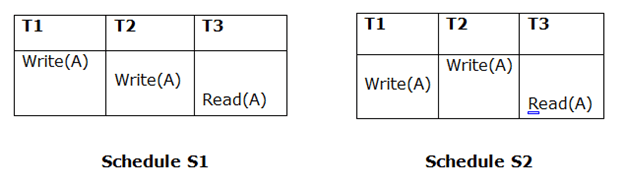
🡺Two schedules S1 and S2 are said to be view equivalent if they satisfy the following conditions:

* **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.



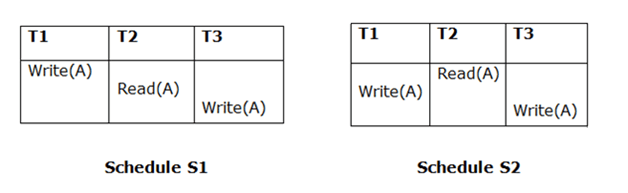
Above two schedules are view equivalent because Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

* **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.

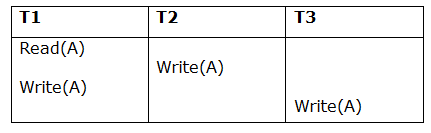


Above two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

* **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

= 3! = 6

S1 = <T1 T2 T3>

S2 = <T1 T3 T2>

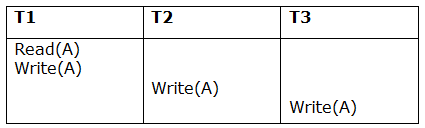
S3 = <T2 T3 T1>

S4 = <T2 T1 T3>

S5 = <T3 T1 T2>

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.

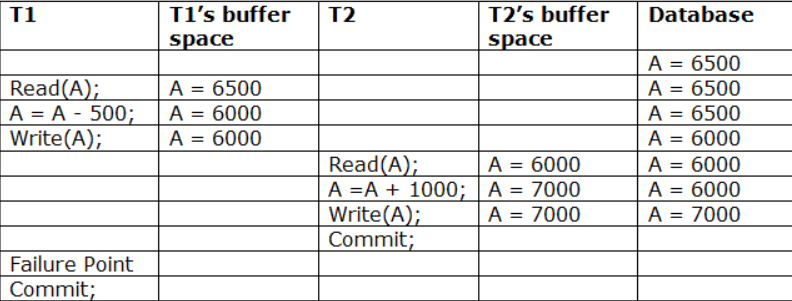
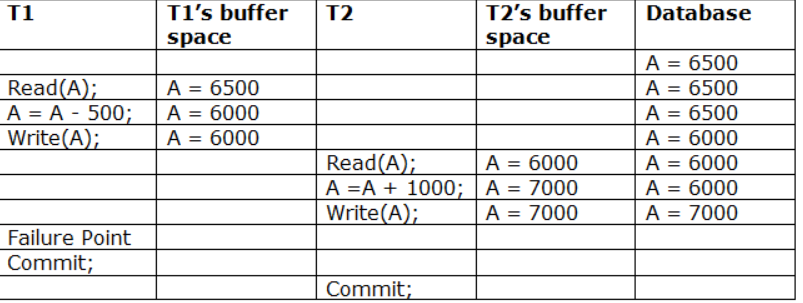
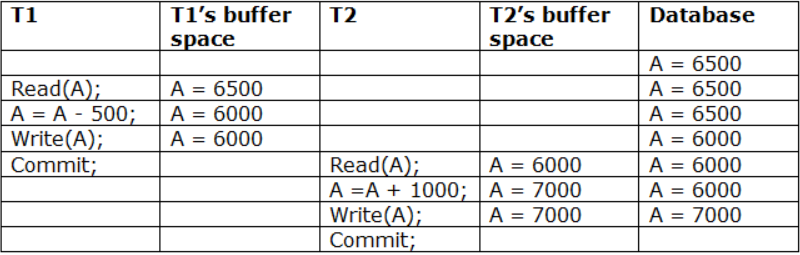
**Step 3:** Final Write

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.

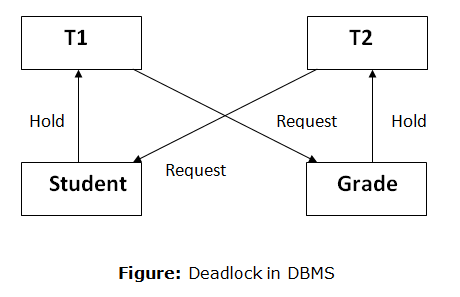
**Hence, view equivalent serial schedule is:**

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.
* **Deadlock handling and distributed database**

**(deadlock)**

* 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.



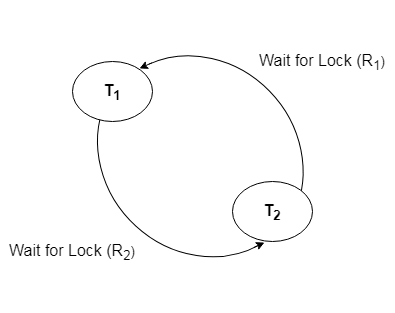
* Below is a list of conditions necessary for a deadlock to occur:
  1. **Circular Waiting:**
* It is when two or more transactions wait each other indefinitely for a lock held by the others to be released.
  1. **Partial Allocation:**
* When a transaction acquires some of the required data items but not all the data items as they may be exclusively locked by others.
  1. **Non-Preemptive scheduling:**
* A data item that could be only single transaction at a time.
  1. **Mutual Exclusion:**
* A data item can be locked exclusively by one transaction at a time.
* **Deadlock Avoidance**
  + 1. 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.
    2. 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.



## 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.

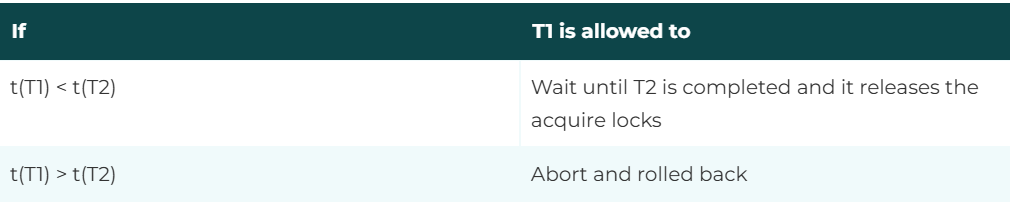
## The Database management system analyzes the operations of the transaction whether they can create a deadlock situation or not. If they do, then the DBMS never allowed that transaction to be executed.

## Each transaction has unique identifier which is called timestamp. It is usually based on the state of the transaction and assigned once the transaction is started. For example if the transaction T1 starts before the transaction T2 then the timestamp corresponding to the transaction T1 will be less than timestamp corresponding to transaction T2. The timestamp decides whether a transaction should wait or abort and rollback. Aborted transaction retain their timestamps values and hence the seniority.

* **The following deadlock prevention schemes using timestamps have been proposed.**
  + - 1. Wait-Die scheme
      2. Wound wait scheme
* The significant disadvantage of both of these techniques is that some transactions are aborted and restarted unnecessarily even though those transactions never actually cause a deadlock.

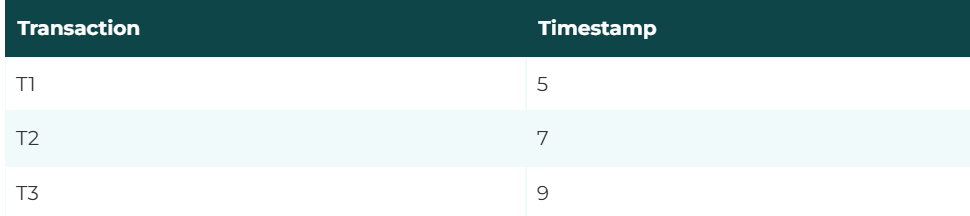
### **Wait-Die scheme**

* 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.



In the above representation, T1 is requesting transactions and T2 is the transaction holding the lock on data item and t(Ti) is the timestamp of the transaction Ti.

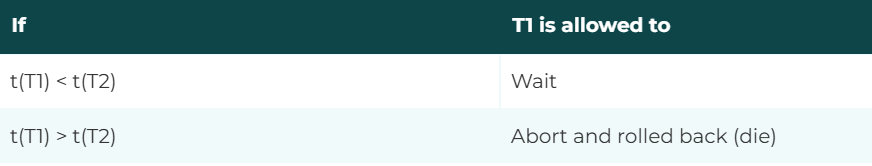
**Consider an example in which transactions having following timestamps:**



If T1 request a data item is locked by transaction T2 then T1 has to wait until T2 completes and all locks acquired by it are released because t(T1) < t(T2). On the other hand, if transaction T3 requests a data item locked by transaction T2 and T3 has to abort and rollback i.e. dies because t(T3) < t(T2).

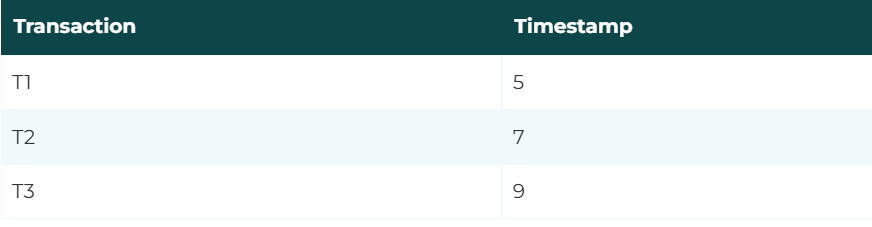
### **Wound wait scheme**

* 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.
* It is based on preemptive technique.



In the above representation, T1 is the requesting transactions and T2 is the transaction holding the lock on data item and t(Ti) is the timestamp of the transaction Ti.

**Consider an example in which transactions having following timestamps:**



If T1 request a data item is locked by transaction T2 then T1 has to wait until T2 completes and all locks acquired by it are released because t(T1) < t(T2). On the other hand, if transaction T3 requests a data item locked by transaction T2 then T3 has to wait because t(T3) > t(T2).

* **Deadlock Detection and Recovery**
* In the deadlock detection scheme, the deadlock detection algorithm checks the state of the system periodically whether the deadlock has occurred or not, if the deadlock exists in the system tries to recover from the deadlock.
* **In order to detect a deadlock the system must have the following information:**
  1. The system should have information about the concurrent group of transactions.
  2. Information about the current allocation of data items for each transaction.
  3. It must have information about the current set of data items that each transaction is waiting for.

The system must provide an algorithm that uses this information i.e. the information about the current allocations of data items to examine whether a system has entered a deadlock state or not. If the deadlock exists then the system attempts to recover from the deadlock.

* **Recovery from Deadlock**
* If the wait for graph which is used for deadlock detection contains a deadlock situation i.e. there exists cycles in it then those cycles should be removed to recover from the deadlock. The most widely used technique of recovering from a deadlock is to rollback one or more transactions till the system no longer displays a deadlock condition.
* **The selection of the transactions to be rolled back is based on the following deliberations:**

1. **Selection of victim:**

* There may be many transactions which are involved in a deadlock i..e deadlocked transaction. So to recover from the deadlock some of the transaction should be rolled back, out of the possible transactions causing a deadlock. The one that is rolled back is known as victim transaction and the mechanism is known as victim election.
* The transactions to be rolled back are the one which has just started or has not made many changes. Avoid selecting transactions that have made many updates and have been running for a long time.

1. **Rollback:**

* Once the selection of the transaction to be rolled back is decided we should find out how far the current transaction should be rolled back. One of the simplest solution is the total rollback i.e. abort the transaction and restart it. However, the transaction should be rolled back to the extent required to break the deadlock. Also, the additional information of the state of currently executing transactions should be maintained.

1. **Starvation:**

* To recover from the deadlock, we must ensure that the same transaction should not be selected again and again as a victim to rollback. The transaction will never complete if the type of situation is not avoided. To avoid starvation, only a finite number of times a transaction should be picked up as a victim.
* A widely used solution is to include the number of rollbacks of the transaction that is selected as the victim.
* **Concurrency control**
* Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.
* But before knowing about concurrency control, we should know about concurrent execution.
* **Concurrent Execution in DBMS**
* In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
* While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
* The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

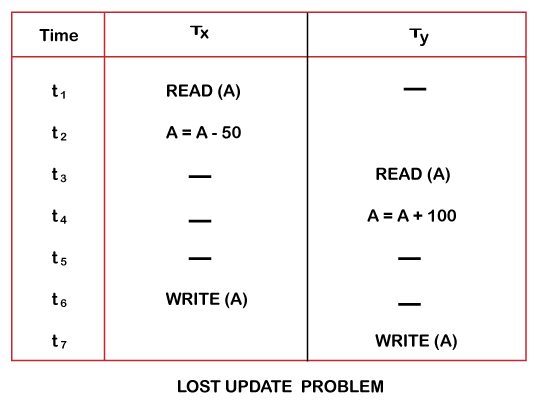
## Problems with Concurrent Execution

* In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

**Problem 1: Lost Update Problems (W - W Conflict)**

* *The problem occurs*when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent*.*
* **For example:**

**Consider the below diagram where two transactions TX and TY, are performed on the same account A where the balance of account A is $300.**



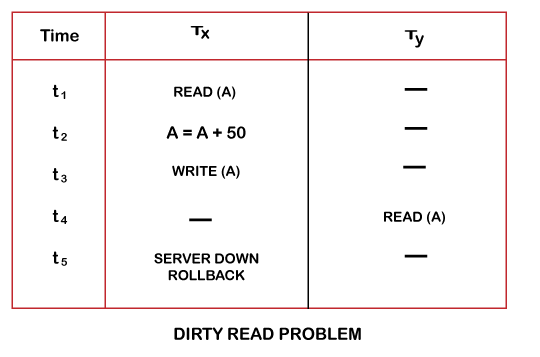
* At time t1, transaction TX reads the value of account A, i.e., $300 (only read).
* At time t2, transaction TX deducts $50 from account A that becomes $250 (only deducted and not updated/write).
* Alternately, at time t3, transaction TY reads the value of account A that will be $300 only because TX didn't update the value yet.
* At time t4, transaction TY adds $100 to account A that becomes $400 (only added but not updated/write).
* At time t6, transaction TX writes the value of account A that will be updated as $250 only, as TY didn't update the value yet.
* Similarly, at time t7, transaction TY writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by TX is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

### **Problem 2: Dirty Read Problems (W-R Conflict)**

* *The* dirty *read* problem *occurs*when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.
* **For example:**

**Consider two transactions TX and TY in the below diagram performing read/write operations on account A where the available balance in account A is $300:**

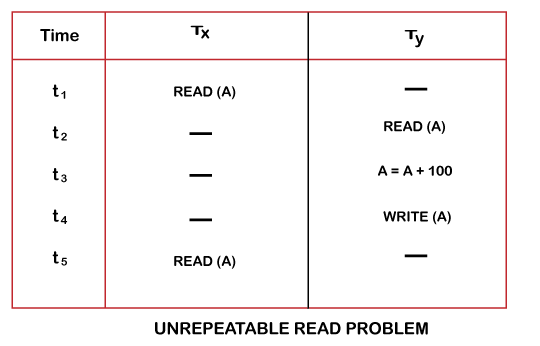


* At time t1, transaction TX reads the value of account A, i.e., $300.
* At time t2, transaction TX adds $50 to account A that becomes $350.
* At time t3, transaction TX writes the updated value in account A, i.e., $350.
* Then at time t4, transaction TY reads account A that will be read as $350.
* Then at time t5, transaction TX rollbacks due to server problem, and the value changes back to $300 (as initially).
* But the value for account A remains $350 for transaction TY as committed, which is the dirty read and therefore known as the Dirty Read Problem.

### **Problem 3:Unrepeatable Read Problem (W-R Conflict)**

* Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.
* **For example:**

**Consider two transactions, TX and TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:**



* At time t1, transaction TX reads the value from account A, i.e., $300.
* At time t2, transaction TY reads the value from account A, i.e., $300.
* At time t3, transaction TY updates the value of account A by adding $100 to the available balance, and then it becomes $400.
* At time t4, transaction TY writes the updated value, i.e., $400.
* After that, at time t5, transaction TX reads the available value of account A, and that will be read as $400.
* It means that within the same transaction TX, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction TY, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

## Concurrency Control

* Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.
* Concurrency Control Protocols
* The concurrency control protocols ensure the atomicity, consistency, isolation, durability and serializability of the concurrent execution of the database transactions.
* Therefore, these protocols are categorized as:
  + 1. **Lock Based Concurrency Control Protocol**
    2. **Time Stamp Concurrency Control Protocol**
    3. **Validation Based Concurrency Control Protocol**
    4. **Multiversion concurrency control protocol**

1. **Lock based concurrency control protocol**

* It is associated with each data item that represents the state of the item with respect to the possible operations that can be performed on it. Its value is used in a locking scheme to manipulation of the associated data item and control concurrent access. Locking a data item being used by a transaction can prevent other transactions running simultaneously from using these locked data items. This is one of the most commonly used techniques to ensure serialization, manipulation of the value of a lock is called locking.
* **Types of Locks**
* Various types of locks are used to control concurrency. Depending on the type of lock, the lock manager grants or denies access to other operations on the same data item.
* **Binary locks**

**A binary lock can have only two values:**

* **Locked**
* **Unlocked**

Which are represented by 1 or 0 for simplicity. Each data item has a separate lock associated with it. If the data item is locked then it cannot be accessed by database operations that request the data item and if the data item is unlocked then it can be accessed when requested.

**Following two operations are associated with binary locking of a data item A.**

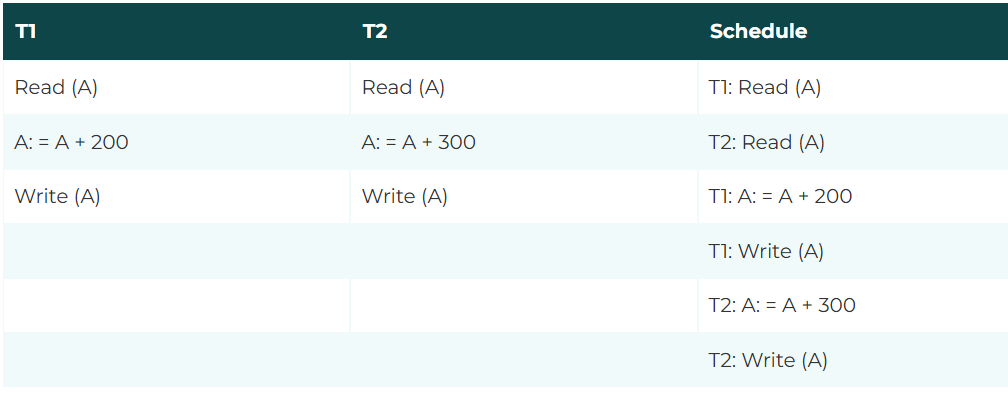
* Lock (A)
* Unlock (A)

A transaction requests access to a data item by first locking the data item using the Lock() operation. While doing so, if another operation of another concurrent transaction tries to access the same data item, it is forced to wait until the transaction that locked the data item has unlocked the same data item using the Unlock() operation. When a given transaction has locked a data item completes all its operations with that data item, it automatically unlocks the data item so that other transactions can use it.

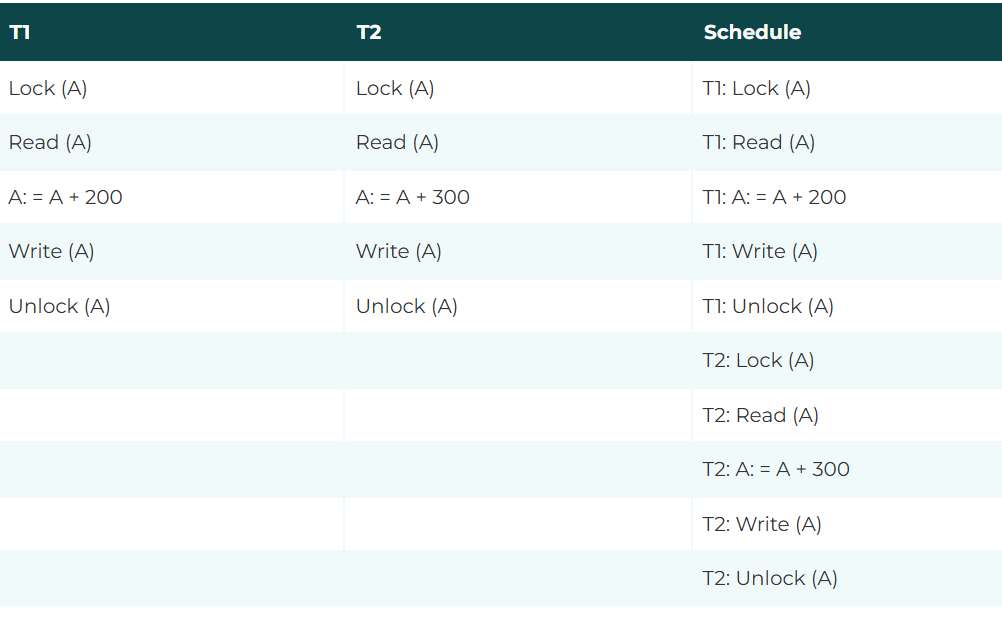
**The following rules must be followed whenever binary locking schemes are used:**

* **Lock ():** This operation must be issued by the transaction before any update operations such as a read or write performed on the transaction.
* **Unlock ():** This operation must be issued by the transaction after all read or write operations in the transaction have completed.
* A lock() operation cannot be released by a transaction if it already holds a lock on the data item.
* An Unlock () operation cannot be issued by the transaction unless it already holds a lock on the data item.

Consider two transactions T1 and T2 both update the account balance by Rs 200 and Rs 300 respectively and the possible schedules is shown if these transactions are made to run concurrently.



Thus the lost update problem occurs in this schedule. Now if we apply binary lock on the transactions then this will happen.



Thus it is clear that a schedule consisting of transactions T1 and T2 will be serializable. This is because if T1 locks account A before T2 does then T2 cannot proceed until the transaction T1 unlocks the account A.

Similarly, if T2 locks account A before T1 does then T1 cannot proceed further until the transactions T2 unlocks the account A.

In binary locking method at most one of the transactions can hold a lock on a particular data item. Thus, no two transactions can access the same item.

* **There are two types of lock:**

**1. Shared lock:**

* It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
* It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

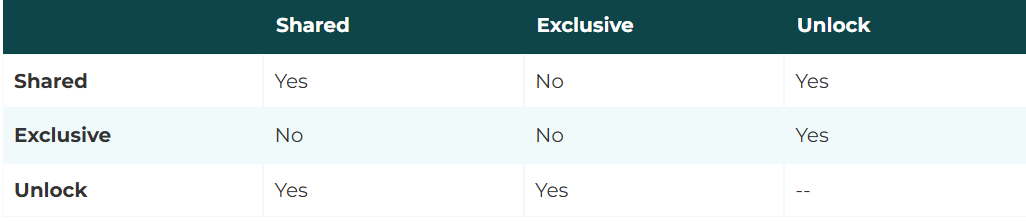
**2. Exclusive lock:**

* In the exclusive lock, the data item can be both reads as well as written by the transaction.
* This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

**The following rules must be followed whenever shared/exclusive locks are used:**

* A shared/exclusive lock must be applied on a data item in a transaction before a read operation is performed in the transaction.
* An exclusive lock must be applied on the data item in a transaction before any write operation on the data item in the transaction have completed.
* A transaction must issue an unlock operation on a data item after all read/write operations on the data item in the transaction have completed.
* A transaction must not execute an unlock operation unless it already holds a shared/exclusive lock on the data item.
* A transaction that already holds a shared/exclusive lock on the data item cannot issue a shared lock on the same data item. Similarly, a transaction that already holds a shared/exclusive lock on the data item cannot issue an exclusive lock on the same data item.

The compatibility relation between shared, exclusive and unlock is given be the Matrix as shown below. In this matrix, we assume that the request for locking is made by the transaction that does not already hold a lock on the data item.

****

**The above matrix can be explained as follows:**

* If the current state of locking on a data item is shared such as a transaction has already applied a share lock on data item X then another transaction can also apply a shared lock on the same data item X i.e. lock mode of request can be shared. But it cannot apply exclusive lock on data item X which is already in shared lock by some other transaction. However you can unlock the data item X that is being applied shared locked by some transaction.
* If the current state of locking on a data item is exclusive i.e. a transaction has already issued an exclusive lock on data item X then other transaction cannot issue a shared or exclusive lock on the same data item X. However, we can unlock the data item X which is being locked exclusively by the same transaction.
* If the data item X is not locked i.e. it is unlocked then it can be locked using shared/ exclusive lock. However, an unlocked data item cannot be unlocked again.

Now consider the situations where two banking accounts A and B exist and let their balances be Rs 1000 and Rs 900. Let us take two transactions creating balance in accounts and another transaction transfers Rs. 200 from account A to account B.



If the above two transactions T1 and T2 are executed in a serial order like T1 and then T2 or T2 and then T1 then the transaction T2 will display a value of Rs 1900 each time. Here Lock X means exclusive lock and Lock S means shared lock. If T1 and T2 transactions are performed simultaneously then schedule as shown below.

In the above schedule, the value of the sum of the balances of accounts A and B comes out to be Rs 2100 which is incorrect. This locking scheme did not solve the inconsistent read problem because after these transactions were executing simultaneously, an inconsistent state of the database was created because the value of account A was added to the 'sum' before the modifications were made to the balance of account A.

This schedule is a non-serializable schedule and a non-serializable schedule occurred even though both the transactions locked and unlocked the data items they used. Thus, locking can help in achieving serializibilty but it does not guarantee it.

## There are four types of lock protocols available:

### 1. Simplistic lock protocol

🡺It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

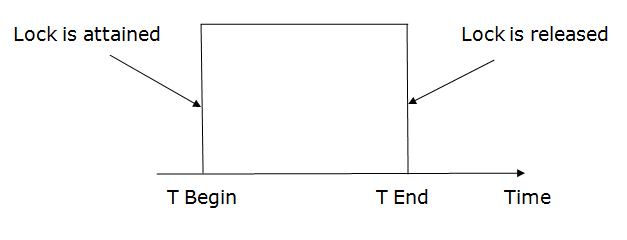
### 2. Pre-claiming Lock Protocol

🡺Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.

🡺Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.

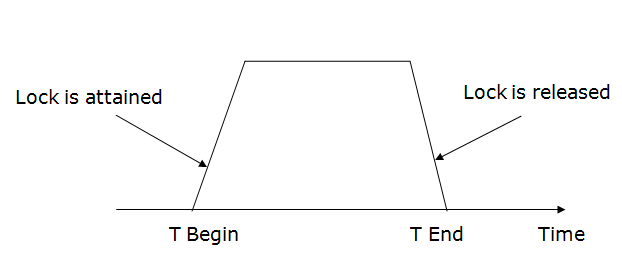
🡺If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.

🡺If all the locks are not granted then this protocol allows the transaction to rolls back and waits until all the locks are granted.



### 3. Two-phase locking (2PL)

* The two-phase locking protocol divides the execution phase of the transaction into three parts.
* In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
* In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
* In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

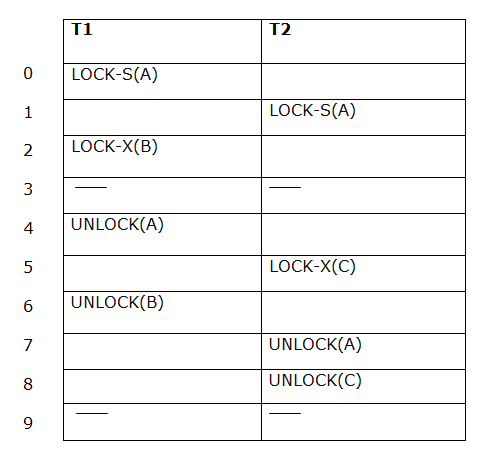
**Growing phase:** In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

**Shrinking phase:** In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

**Example:**



The following way shows how unlocking and locking work with 2-PL.

**Transaction T1:**

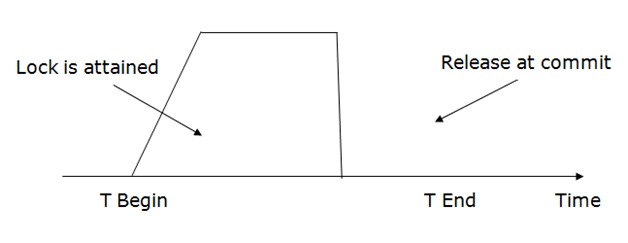
* **Growing phase:** from step 1-3
* **Shrinking phase:** from step 5-7
* **Lock point:** at 3

**Transaction T2:**

* **Growing phase:** from step 2-6
* **Shrinking phase:** from step 8-9
* **Lock point:** at 6

### 4. Strict Two-phase locking (Strict-2PL)

* The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
* The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
* Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
* Strict-2PL protocol does not have shrinking phase of lock release.



It does not have cascading abort as 2PL does.

1. **Timestamp based protocol**

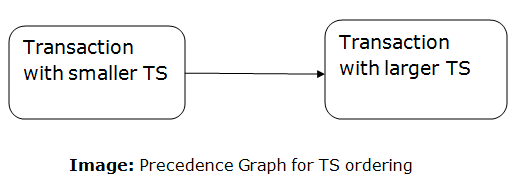
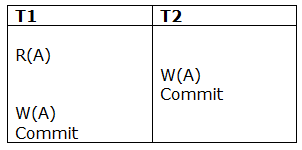
* The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
* The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
* The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
* Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
* The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.
* Basic Timestamp ordering protocol works as follows:
  1. Check the following condition whenever a transaction Ti issues a Read (X) operation:
* If W\_TS(X) >TS(Ti) then the operation is rejected.
* If W\_TS(X) <= TS(Ti) then the operation is executed.
* Timestamps of all the data items are updated.
  1. Check the following condition whenever a transaction Ti issues a Write(X) operation:
* If TS(Ti) < R\_TS(X) then the operation is rejected.
* If TS(Ti) < W\_TS(X) then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Where,

TS(TI) denotes the timestamp of the transaction Ti.

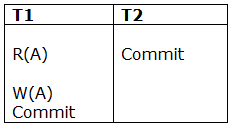
R\_TS(X) denotes the Read time-stamp of data-item X.

W\_TS(X) denotes the Write time-stamp of data-item X.

* Advantages and Disadvantages of TO protocol:
* TO protocol ensures serializability since the precedence graph is as follows:  
  
* TS protocol ensures freedom from deadlock that means no transaction ever waits.
* But the schedule may not be recoverable and may not even be cascade- free.
* **Thomos’write rule**
* Thomas Write Rule provides the guarantee of serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.
* The basic Thomas write rules are as follows:
* If TS(T) < R\_TS(X) then transaction T is aborted and rolled back, and operation is rejected.
* If TS(T) < W\_TS(X) then don't execute the W\_item(X) operation of the transaction and continue processing.
* If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set W\_TS(X) to TS(T).
* If we use the Thomas write rule then some serializable schedule can be permitted that does not conflict serializable as illustrate by the schedule in a given figure:  
  
* Figure: A Serializable Schedule that is not Conflict Serializable

In the above figure, T1's read and precedes T1's write of the same data item. This schedule does not conflict serializable.

Thomas write rule checks that T2's write is never seen by any transaction. If we delete the write operation in transaction T2, then conflict serializable schedule can be obtained which is shown in below figure.



* Figure: A Conflict Serializable Schedule

1. **Validation Based Concurrency Control Protocol**

* Validation phase is also known as optimistic concurrency control technique. In this technique, it is assumed that all data items can be successfully updated at the end of a transaction.
* While applying all updates, they are applied to the local copies of the data item that are kept for the transaction. At the end, the validation phase check whether any of the transactions updates violates serializiblity, if serializiblity is not violated the commit operation is performed and database is updated from local copies. Otherwise transaction will be rolled back and restarted.
* Optimistic scheduling methods are called optimistic because they assume that there will be little interference and hence there is no need to check during transaction execution.

1. Read phase: In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
2. Validation phase: In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. Write phase: If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

* The validation phase examines the reads and writes of the transaction that may cause overlapping. So each transaction is assigned with the following different timestamps:

Start(Ti): It contains the time when Ti started its execution.

Validation (Ti): It contains the time when Ti finishes its read phase and starts its validation phase.

Finish(Ti): It contains the time when Ti finishes its write phase.

* This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.
* Hence TS(T) = validation(T).
* The serializability is determined during the validation process. It can't be decided in advance.
* While executing the transaction, it ensures a greater degree of concurrency and also less number of conflicts.
* Thus it contains transactions which have less number of rollbacks.

A transaction T can complete its validation phase successfully if at least one of the following conditions is satisfied:

* All older transaction i.e. the transactions with smaller timestamps must have completed before the requesting transaction started.
* If a transaction T starts before earlier one finishes then the transaction T should not read the data items written by the earlier transactions. This rule guarantees that write of earlier transactions are not read by other transaction T.
* If a transaction T starts before earlier one finishes then the earlier transaction should complete its write phase before transaction T enters its validation phase. So this rule guarantees that writes are done serially ensuring that there is no conflict.

When validating a transaction T, the first condition is checked for the preceding transaction. If it is false, only condition 2 is checked. If condition 2 is false, only condition 3 is checked. If any one of these three conditions evaluates to true, there is no conflict and T is successfully validated. The validation phase of transaction T fails if none of these three conditions are true and it is then rolled back and restarted later.

* The main idea for using validation based protocol is:
* Minimum Overhead: All the data items are updated at the end of the transaction so minimum overhead is caused during the execution of transaction.
* No cascade rollback: Since the rollbacks involve only a local copy of data and no database is involved, so there will not be any cascading rollbacks.
* No locking required: This technique allows greater concurrency then traditional protocols since no locking is required.
* Efficient: This technique is very efficient when conflicts are rare. Occasionally conflicts results in transaction rollbacks.
* The main disadvantage of using validation based protocols is:
* If the rolled back transaction is very long then valuable processing time will be lost.
* These techniques do not work well when there is a lot of interference between transactions because the results of the transaction that is committed to completion will be discarded and must be restarted.

1. **Multiversion concurrency control**

* Multi-version protocol aims to reduce the delay for read operations. It maintains multiple versions of data items.
* Whenever a write operation is performed, the protocol creates a new version of the transaction data to ensure conflict-free and successful read operations.
* The newly created version contains the following information –

1. Content − This field contains the data value of that version.
2. Write\_timestamp − This field contains the timestamp of the transaction that created the new version.
3. Read\_timestamp − This field contains the timestamp of the transaction that will read the newly created value.

* By creating multiple versions of the data, the multi-version protocol ensures that read operations can access the appropriate version of the data without encountering conflicts. The protocol thus enables efficient concurrency control and reduces delays in read operations.
* Various Types of MVCC

|  |  |  |  |
| --- | --- | --- | --- |
| MVCC Type | Description | Advantages | Disadvantages |
| Snapshot-based | Creates a snapshot of the database at the start of a transaction and uses it to provide necessary data for the transaction | Easy to implement | Significant overhead due to storing multiple versions of data |
| Timestamp-based | Assigns a unique timestamp to each transaction that creates a new version of a record; used to determine data visibility to transactions | More efficient than snapshot-based MVCC | Requires additional storage to store timestamps |
| History-based | Stores a complete history of all changes made to a record, allowing for easy rollback of transactions | Provides highest level of data consistency | Most complex of the MVCC techniques |
| Hybrid | Combines two or more MVCC techniques to balance performance and data consistency | Provides benefits of multiple MVCC techniques | More complex to implement than individual techniques |

## Benefits of multiversion concurrency control (MVCC)

### Less need for database locks

* With MVCC, the database can allow multiple transactions to read and write data without locking the entire database.

### Fewer issues with multiple transactions trying to access the same data

* MVCC helps reduce conflicts between transactions accessing the same data.

### Faster access to read data

* Since MVCC allows multiple transactions to read data at the same time, it improves the speed of reading data.

### Records are still protected during write operations

* MVCC ensures that data is protected from being changed by other transactions while a transaction is making changes to it.

### Fewer database deadlocks

* Deadlocks occur when two or more transactions are waiting for each other to release a lock, causing the system to come to a halt. MVCC can reduce the number of these occurrences.

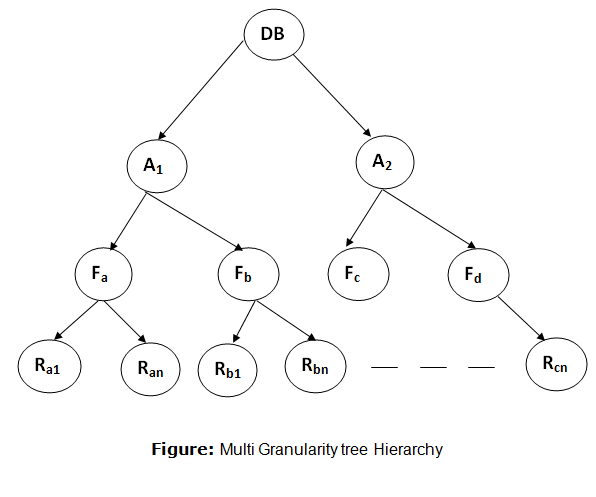
## Drawbacks of Multiversion concurrency control (MVCC)

## Concurrent update control methods can be challenging to implement.

## The database can become bloated with multiple versions of records, which increases its overall size.

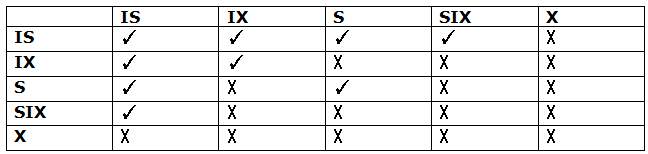
* **Multiple granularity**
* **Granularity:**
* It is the size of data item allowed to lock.

### **Multiple Granularity:**

* It can be defined as hierarchically breaking up the database into blocks which can be locked.
* The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
* It maintains the track of what to lock and how to lock.
* It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.
* **For example:** Consider a tree which has four levels of nodes.
* The first level or higher level shows the entire database.
* The second level represents a node of type area. The higher level database consists of exactly these areas.
* The area consists of children nodes which are known as files. No file can be present in more than one area.
* Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
* Hence, the levels of the tree starting from the top level are as follows:
  1. Database
  2. Area
  3. File
  4. Record
* 
* There are three additional lock modes with multiple granularity:

**Intention Mode Lock**

* Intention-shared (IS): It contains explicit locking at a lower level of the tree but only with shared locks.
* Intention-Exclusive (IX): It contains explicit locking at a lower level with exclusive or shared locks.
* Shared & Intention-Exclusive (SIX): In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.
* Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:



It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

* Transaction T1 should follow the lock-compatibility matrix.
* Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
* If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
* If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
* If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
* If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

* If transaction T1 reads record Ra9 in file Fa, then transaction T1 needs to lock the database, area A1 and file Fa in IX mode. Finally, it needs to lock Ra2 in S mode.
* If transaction T2 modifies record Ra9 in file Fa, then it can do so after locking the database, area A1 and file Fa in IX mode. Finally, it needs to lock the Ra9 in X mode.
* If transaction T3 reads all the records in file Fa, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock Fa in S mode.
* If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.