Unit - 9: Transaction and Concurrency Control

Transaction Concept

The term **transaction** refers to a collection of operations that form a single logical unit of work. For instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

Collections of operations that form a single logical unit of work are called **transactions**. A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does. Furthermore, it must manage concurrent execution of transactions in a way that avoids the introduction of inconsistency

A transaction is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language (typically SQL), or programming language (for example, C++, or Java), with embedded database accesses in JDBC or ODBC.

Let's take an example of a simple transaction. Suppose a bank employee transfers Rs 500 from A's account to B's account. This very simple and small transaction involves several low-level tasks.

A's Account

```
Open_Account(A)
Old_Balance = A.balance
New_Balance = Old_Balance - 500
A.balance = New_Balance
Close_Account(A)
```

B's Account

```
Open_Account(B)
Old_Balance = B.balance
New_Balance = Old_Balance + 500
B.balance = New_Balance
Close_Account(B)
```

ACID properties

A transaction is a unit of a program execution and it may contain several low level tasks. A transaction in a database system must maintain **A**tomicity, **C**onsistency, **I**solation, and **D**urability − commonly known as ACID properties − in order to ensure accuracy, completeness, and data integrity.

- **Atomicity:** *Either all operations of the transaction are reflected properly in the database, or none are.*

  This property states that a transaction must be treated as an atomic unit, that is, either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed. States should be defined either before the execution of the transaction or after the execution/abortion/failure of the transaction.

  Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

  If the transaction fails after completion of **T1** but before completion of **T2**.(say, after **write(X)** but before **write(Y)**), then amount has been deducted from **X** but not added to **Y**.

  This results in an inconsistent database state.

  Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

- **Consistency:** *Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database*.

  The database must remain in a consistent state after any transaction. No transaction should have any adverse effect on the data residing in the database. If the database was in a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.

  Referring to the example above, The total amount before and after the transaction must be maintained.

  Total **before T** occurs = **500 + 200 = 700**. Total **after T occurs** = **400 + 300 = 700**.

  Therefore, database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result T is incomplete.

- **Isolation:** *Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj , it appears to Ti that either Tj finished execution before Ti started or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.*

  In a database system where more than one transaction are being executed simultaneously, the property of isolation states that all the transactions will be carried out and executed as if it is the only transaction in the system. **No transactionwill affect the existence of any other transaction.**

  Example:

  Let X= 500, Y = 500.

  Consider two transactions T and and T"

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write	Read (X) Read (Y) Z: = X + Y Write (Z)

Suppose **T** has been executed till **Read (Y)** and then **T''** starts. As a result , interleaving of operations takes place due to which **T''** reads correct value of **X** but incorrect value of **Y** and sum computed by

T'': (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of transaction:

T: (X+Y = 50, 000 + 450 = 50, 450).

This results in database **inconsistency**, due to a **loss of 50 units**. Hence, transactions must take place in **isolation** and changes should be visible only after a they have been made to the main memory.

- **Durability:** *After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.*

  The database should be durable enough to hold all its latest updates even if the system fails or restarts. If a transaction updates a chunk of data in a database and commits, then the database will hold the modified data. If a transaction commits but the system fails before the data could be written on to the disk, then that data will be updated once the system springs back into action

These properties are often called the ACID properties; the acronym is derived from the first letter of each of the four properties

To gain a better understanding of ACID properties and the need for them, consider a simplified banking system consisting of several accounts and a set of transactions that access and update those accounts. For the time being, we assume that the database permanently resides on disk, but that some portion of it is temporarily residing in main memory.

Transactions access data using two operations:

read(X): which transfers the data item X from the database to a local buffer belonging to the transaction that executed the read operation.

write(X): which transfers the data item X from the local buffer of the transaction that executed the write back to the database.

Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as

Ti: read(A);

A := A − 50;

write(A);

read(B);

B := B + 50;

write(B).

Let us now consider each of the ACID requirements. (For ease of presentation, we consider them in an order different from the order A-C-I-D).

Consistency: The consistency requirement here is that the sum of *A* and *B* be unchanged by the execution of the transaction. Without the consistency requirement, money could be created or destroyed by the transaction! It can be verified easily that, if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

Atomicity: Suppose that, just before the execution of transaction *Ti* the values of accounts *A* and *B* are $1000 and $2000, respectively. Now suppose that, during the execution of transaction *Ti*, a failure occurs that prevents *Ti* from completing its execution successfully. Examples of such failures include power failures, hardware failures, and software errors. Further, suppose that the failure happened after the write(*A*) operation but before the write(*B*) operation. In this case, the values of accounts *A* and *B* reflected in the database are $950 and $2000. The system destroyed $50 as a result of this failure. In particular, we note that the sum $A + B$ is no longer preserved. Thus, because of the failure, the state of the system no longer reflects a real state of the world that the database is supposed to capture. We term such a state an **inconsistent state**. We must ensure that such inconsistencies are not visible in a database system. Note, however, that the system must at some point be in an inconsistent state. Even if transaction *Ti* is executed to completion, there exists a point at which the value of account *A* is $950 and the value of account *B* is $2000, which is clearly an inconsistent state. This state, however, is eventually replaced by the consistent state where the value of account *A* is $950, and the value of account *B* is $2050. Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction. That is the reason for the atomicity requirement: If the atomicity property is present, all actions of the transaction are reflected in the database, or none are. The basic idea behind ensuring atomicity is this: The database system keeps track (on disk) of the old values of any data on which a transaction performs a write, and, if the transaction does not complete its execution, the database system restores the old values to make it appear as though the transaction never executed.

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds. The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution. We assume for now that a failure of the computer system may result in loss of data in main memory, but data written to disk are never lost. We can guarantee durability by ensuring that either:

- The updates carried out by the transaction have been written to disk before the transaction completes.
- Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.

Isolation: Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way, resulting in an inconsistent state. For example, as we saw earlier, the database is temporarily inconsistent while the transaction to transfer funds from A to B is executing, with the deducted total written to A and the increased total yet to be written to B. If a second concurrently running transaction reads A and B at this intermediate point and computes $A+B$, it will observe an inconsistent value. Furthermore, if this second transaction then performs updates on A and B based on the inconsistent values that it read, the database may be left in an inconsistent state even after both transactions have completed. A way to avoid the problem of concurrently executing transactions is to execute transactions serially—that is, one after the other. However, concurrent execution of transactions provides significant performance benefits. Other solutions have therefore been developed; they allow multiple transactions to execute concurrently

Transaction Atomicity and Durability

- A transaction may not always complete its execution successfully. Such a transaction is termed **aborted**. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database.
- Thus, any changes that the aborted transaction made to the database must be undone. Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**. It is part of the responsibility of the **recovery scheme** to manage transaction aborts. This is done typically by maintaining a log. Each database modification made by a transaction is first recorded in the log. We record the identifier of the transaction performing the modification, the identifier of the data item being modified, and both the old value (prior to modification) and the new value (after modification) of the data item. Only then is the database itself modified. Maintaining a log provides the possibility of redoing a modification to ensure atomicity and durability as well as the possibility of undoing a modification to ensure atomicity in case of a failure during transaction execution.
- A transaction that completes its execution successfully is said to be **committed**. A committed transaction that has performed updates transforms the database into a new consistent state, which must persist even if there is a system failure.
- Once a transaction has committed, we cannot undo its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a compensating transaction. For instance, if a transaction added $20 to an account, the compensating transaction would subtract $20 from the account. However, it is not always possible to create such a compensating transaction.

Transaction State

A transaction in a database must be in one of the following states –

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    - restart the transaction
        - can be done only if no internal logical error
    - kill the transaction
- **Committed** –If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

The state diagram corresponding to a transaction appears in Figure below. We say that a transaction has committed only if it has entered the committed state. Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have **terminated** if has either committed or aborted. A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion. The database system then writes out enough information to disk that, even in the event of a failure, the updates performed by the transaction can be re-created when the system restarts after the failure. When the last of this information is written out, the transaction enters the committed state.

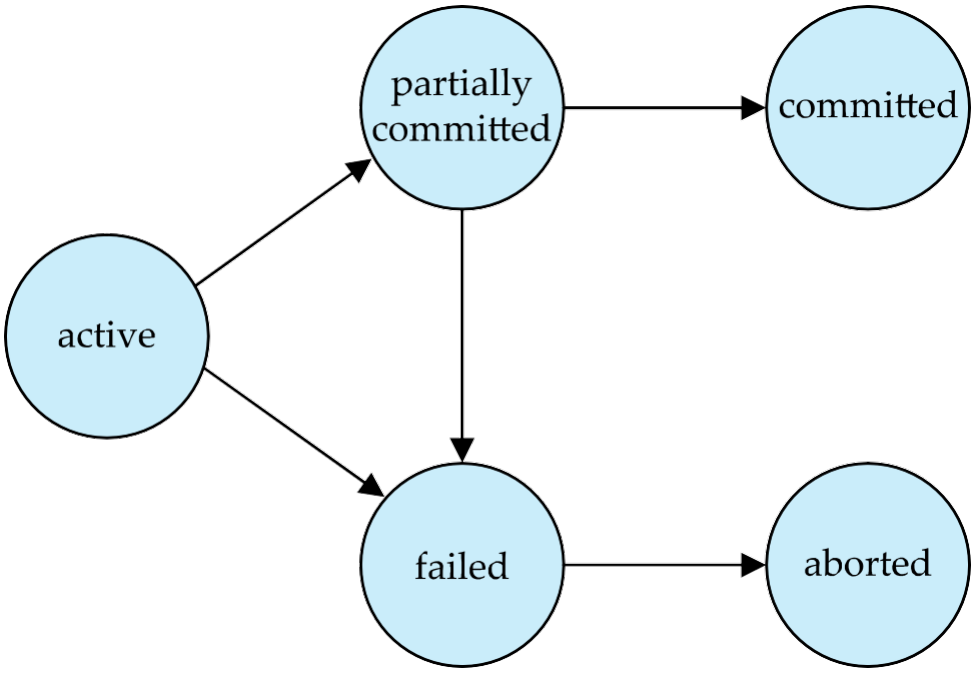


Figure : State diagram of a transaction.

Transaction Isolation

- Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data,

Ensuring consistency in spite of concurrent execution of transactions requires extra work; it is far easier to insist that transactions run **serially**—that is, one at a time, each starting only after the previous one has completed. However, there are two good reasons for allowing concurrency:

- **Improved throughput and resource utilization**: A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. The CPU and the disks in a computer system can operate in parallel. Therefore, I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. While a read or write on behalf of one transaction is in progress on one disk, another transaction can be running in the CPU, while another disk may be executing a read or write on behalf of a third transaction. All of this increases the **throughput** of the system—that is, the number of transactions executed in a

given amount of time. Correspondingly, the processor and disk **utilization** also increase; in other words, the processor and disk spend less time idle, or not performing any useful work.

- **Reduced waiting time**: There may be a mix of transactions running on a system, some short and some long. If transactions run serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to unpredictable delays in running a transaction. If the transactions are operating on different parts of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses among them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the **average response time**: the average time for a transaction to be completed after it has been submitted. The motivation for using concurrent execution in a database is essentially the same as the motivation for using **multiprogramming** in an operating system. When several transactions run concurrently, database consistency can be destroyed despite the correctness of each individual transaction. In this section, we present the concept of schedules to help identify those executions that are guaranteed to ensure consistency.

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
    - **Increased processor and disk utilization**, leading to better transaction *throughput*
        - e.g., one transaction can be using the CPU while another is reading from or writing to the disk
    - **Reduced average response time** for transactions: short transactions need not wait behind long ones.

- **Concurrency control schemes** – mechanisms to achieve isolation
    - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
    - A schedule for a set of transactions must consist of all instructions of those transactions
    - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
    - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

A chronological execution sequence of a transaction is called a **schedule**. A schedule can have many transactions in it, each comprising of a number of instructions/tasks.

Serial and Non Serial Schedule − A schedule in which transactions are aligned in such a way that one transaction is executed first. When the first transaction completes its cycle, then the next transaction is executed. Transactions are ordered one after the other. This type of schedule is called **a serial schedule**, as transactions are executed in a serial manner. If schedule is not serial then it iscalled non –serial schedule.

Examples of serial schedule:

Schedule 1: Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.

Following is a serial schedule in which T_1 is followed by T_2:

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 2: Let T1 transfer \$50 from A to B, and T2 transfer 10% of the balance from A to B. The following is a serial schedule, in which T2 is followed by T1.

T_1	T_2
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	

Example of non-serial schedule:

Schedule 3: Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1

T_1	T_2
read (A)	
$A := A - 50$	
write (A)	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
read (B)	
$B := B + 50$	
write (B)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

In Schedules 1, 2 and 3, the sum A + B is preserved

Schedule 4: The following concurrent schedule does not preserve the value of (A + B).

T_1	T_2
read (A)	
$A := A - 50$	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- When multiple transactions are running concurrently then there is a possibility that the database may be left in an inconsistent state.
- Serializability is a concept that helps us to check which schedules are serializable.
- A serializable schedule is the one that always leaves the database in consistent state.

The main objective of serializability is to search non-serial schedules that allow transaction to execute concurrently without interfering one another transaction and produce the result that could be produced by a serial execution. We concluded that a non-serial schedule is correct if and only if it can produce the same result as that of some serial execution. And such a schedule that is equivalent to the serial schedule is called **serializability**.

In serializability, ordering of read/write is important.

- ✓ *If two transactions only read data item they do not conflict and order is not important.*
- ✓ *If two transactions either read or write completely separate data items, they do not conflict and order is not important.*
- ✓ *If one transaction writes a data item and another reads or writes same data items, then order of execution is important.*

Different forms of schedule equivalence give rise to the notions of:

1. **Conflict serializability**
2. **View serializability**

Simplified view of transactions

- ➢ We ignore operations other than read and write instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instruction.

Conflict Serializability

Instructions li and lj of transactions Ti and Tj respectively, conflict if and only if there exists some item Q accessed by both li and lj, and at least one of these instructions wrote Q.

Let us consider a schedule S in which there are two consecutive instructions Ii and Ij, of transactions Ti and Tj , respectively (i!-= j). If Ii and Ij refer to different data items, then we can swap Ii and Ij without affecting the results of any instruction in the schedule. However, if Ii and Ij refer to the same data item Q, then the order of the two steps may matter. Since we are dealing with only read and write instructions, there are four cases that we need to considered:

1. Ii = read(Q), Ij = read(Q). The order of Ii and Ij does not matter, since the same value of Q is read by Ti and Tj , regardless of the order.
2. Ii = read(Q), Ij = write(Q). If Ii comes before Ij, then Ti does not read the value of Q that is written by Tj in instruction Ij. If Ij comes before Ii, then Ti reads the value of Q that is written by Tj. Thus, the order of Ii and Ij matters.
3. Ii = write(Q), Ij = read(Q). The order of Ii and Ij matters for reasons similar to those of the previous case.

4. Ii = write(Q), Ij = write(Q). Since both instructions are write operations, the order of these instructions does not affect either Ti or Tj . However, the value obtained by the next read(Q) instruction of S is affected, since the result of only the latter of the two write instructions is preserved in the database. If there is no other write(Q) instruction after Ii and Ij in S, then the order of Ii and Ij directly affects the final value of Q in the database state that results from schedule S.

*We say that Ii and Ij **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.*

Conditions for conflicting instructions

- **Different transactions**
- **Same data item**
- **At least one write operation**

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.

We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

The Schedule given below can be transformed into Schedule 1, a serial schedule where T2 follows T1, by a series of swaps of non-conflicting instructions. Therefore this Schedule is **conflict serializable.**

T1	T2
read(A) write(A)	
	Read(A) write(A)
Read(B) write(B)	
	Read(B) write(B)

T1	T2
read(A) write(A) Read(A) write(A)	
	Read(B) write(B) Read(B) write(B)

Schedule 1

Lets take another example:

T1	T2
R(A)	
	R(A)
	R(B)
	W(B)
R(B)	
W(A)	

Lets **swap non-conflicting operations**:

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

T1	T2
	R(A)
R(A)	
	R(B)
	W(B)
R(B)	
W(A)	

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

T1	T2
	R(A)
	R(B)
R(A)	
	W(B)
R(B)	
W(A)	

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

T1	T2
	R(A)
	R(B)
	W(B)
R(A)	
R(B)	
W(A)	

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

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q,*

    1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q*.

    2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .

    3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'*.

- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.

The premise behind view equivalence:

- As long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results.
- "**The view":** the read operations are said to see the same view in both schedules

- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

This schedule is view serializable since it is equivalent to serial schedule <T27,T28,T29>. Here the one read(Q) instruction reads the initial value of Q in both schedules and T29 performs the final write of Q in both schedules. But this schedule is not conflict serailizable since T28 and T29 perform Write(Q) operation without having performed a read(Q) operation. Writes of this sort are called ***blind writes.***

Test for Conflict Serializability

Consider a schedule S. We construct a directed graph, **called a precedence graph**, from S. This graph consists of a pair **G = (V, E),** where **V** is a set of vertices and **E** is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges **Ti →Tj** for which one of three conditions holds:

1. *Ti executes write(Q) before Tj executes read(Q).*
2. *Ti executes read(Q) before Tj executes write(Q).*
3. *Ti executes write(Q) before Tj executes write(Q).*

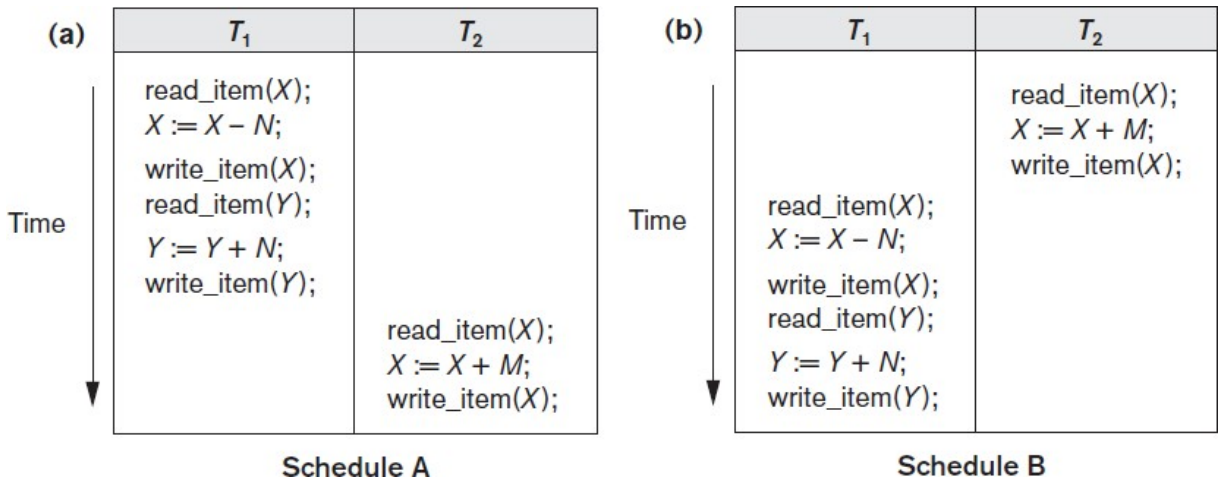
If an edge **Ti → Tj** exists in the precedence graph, then, in any serial schedule S' equivalent to S, Ti must appear before Tj .

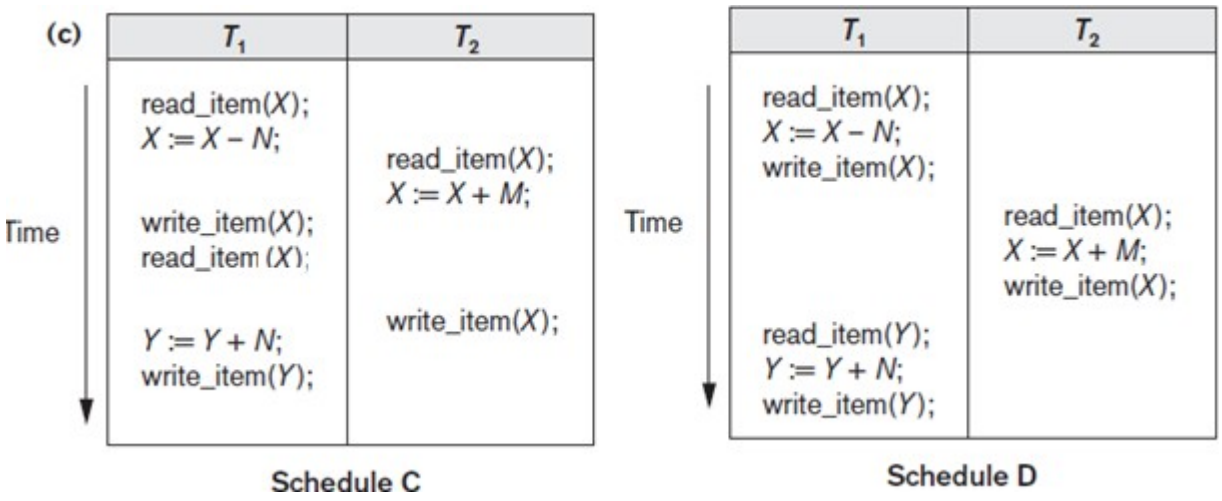
- **A schedule is conflict serializable if and only if its precedence graph is acyclic**.
- **If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.**

Algorithm: Testing Conflict Serializability of a Schedule S

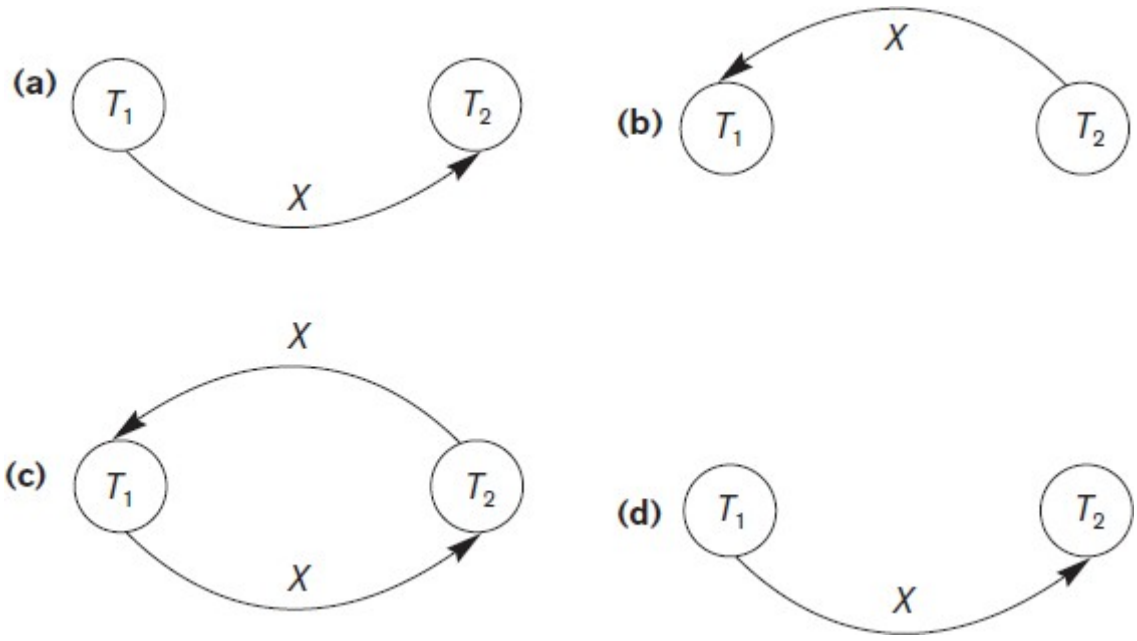
1. For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
2. For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create an edge (Ti→ Tj) in the precedence graph.
3. For each case in S where Tj executes a write_item(X) after Ti executes a read_item(X), create an edge (Ti→Tj) in the precedence graph.
4. For each case in S where Tj executes a write_item(X) after Ti executes a write_item(X), create an edge (Ti→ Tj) in the precedence graph.
5. The schedule S is ***serializable if and only if the precedence graph has no cycles***.

Example:





Constructing the precedence graphs for schedules A to D from above fig. to test for conflict serializability.



(a) Precedence graph for serial schedule A.

(b) Precedence graph for serial schedule B.

(c) Precedence graph for schedule C (not serializable).

(d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Example1: consider the following schedule S: r3(A); r2(A); w3(A); w1(A); r1(A); w1(A)

a) Give the precedence graph P(S)

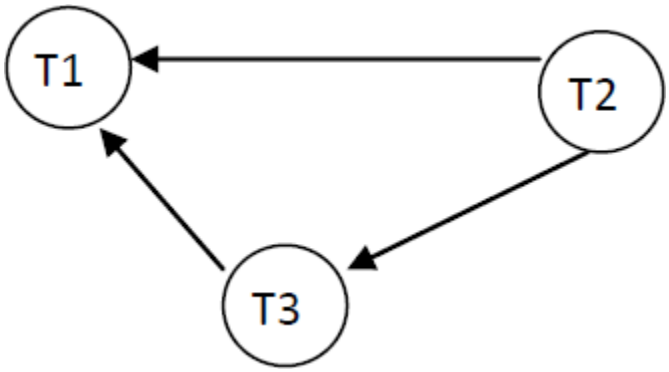
b) Is the schedule conflict serializable. If so, give all equivalent serial schedules

Solution:

Schedule :

T1	T2	T3
		r(A)
	r(A)	
		w(A)
r(A)		
w(A)		

Precedence graph :



Since the above precedence graph is not a cyclic hence given schedule is conflict serializable. And its equivalent serial schedule is T2, T3,T1.

T2	T3	T1
r(A)		
	r(A)	
	w(A)	
		r(A)
		w(A)

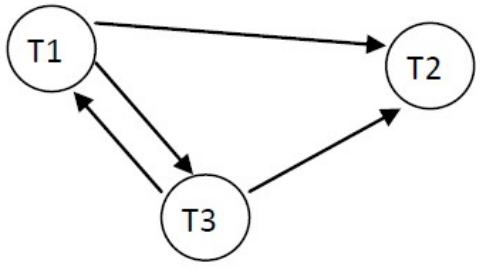
Example 2: consider the following schedule S: w1(A); r2(A); w3(B); w1(B); w3(B); w2(A); r3(B); r2(B)

a) Give the precedence graph P(s)

b) Is the schedule conflict serializable. If so, give all equivalent serial schedules.

Solution:

T1	T2	T3
w(A)		
	r(A)	
		w(B)
w(B)		
		w(B)
	w(A)	
		r(B)
	r(B)	



Dependence graph

Since the above precedence graph is cyclic hence given schedule is not conflict serializable. And its serial schedule cannot be determined.

Transaction Isolation and Atomicity

- There is effect of transaction failures during concurrent execution.
- Need to address the effect of transaction failures on concurrently running transactions.
- If a transaction Ti fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction.
- In a system that allows concurrent execution, the atomicity property requires that any transaction Tj that is dependent on Ti (that is, Tj has read data written by Ti) is also aborted.
- To achieve this, we need to place restrictions on the type of schedules permitted in the system.

Recoverable schedule

- A recoverable schedule is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i appears before the commit operation of T_j.
- The following schedule is not recoverable

T_6	T_7
read(A)	
write(A)	
	read(A)
	commit
read(B)	

- For this should to be recoverable, T_7 would have to delay committing until after T_6 commits

Cascadeless Schedules

— Even if a schedule is recoverable, to recover correctly from the failure of a transaction Ti , we may have to roll back several transactions. Such situations occur if transactions have read data written by Ti .

— In following example, transaction T8 writes a value of A that is read by transaction T9. Transaction T9 writes a value of A that is read by transaction T10.

— Suppose that, at this point, T8 fails. T8 must be rolled back. Since T9 is dependent on T8, T9 must be rolled back. Since T10 is dependent on T9, T10 must be rolled back.

T_8	T_9	T_{10}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)
abort		

— This phenomenon, in which a single transaction failure leads to a series of transaction rollbacks, is called **cascading rollback**.

— Cascading rollback is undesirable, since it leads to the undoing of a significant amount of work.

— It is desirable to restrict the schedules to those where cascading rollbacks cannot occur. Such schedules are called **cascadeless schedules**

— Formally, **a cascadeless schedule** is one where, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti , the commit operation of Ti appears before the read operation of Tj .

— .Every cascadeless schedule is also recoverable

Transaction Isolation Levels

The isolation levels specified by the SQL standard are as follows:

- **Serializable** usually ensures serializable execution. However, some database systems implement this isolation level in a manner that may, in certain cases, allow nonserializable executions.
- **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it. However, the transaction may not be serializable with respect to other transactions. For instance, when it is searching for data satisfying some conditions, a transaction may find some of the data inserted by a committed transaction, but may not find other data inserted by the same transaction.
- **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
- **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.

All the isolation levels above additionally disallow **dirty writes**, that is, they disallow *writes to a data item that has already been written by another transaction that has not yet committed or aborted.*

Concurrency Control

When several transactions execute concurrently in the database, the consistency of data may no longer be preserved. It is necessary for the system to control the interaction among the concurrent transactions, and this control is achieved through one of a variety of mechanisms called **concurrency-control** schemes.

When more than one transactions are running simultaneously there are chances of a conflict to occur which can leave database to an inconsistent state. To handle these conflicts we need concurrency control in DBMS, which allows transactions to run simultaneously but handles them in such a way so that the integrity of data remains intact.

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

There are a variety of concurrency-control schemes. No one scheme is clearly the best; each one has advantages.

Lock-Based Protocols

One way to ensure isolation is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item. The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a **lock** on that item.

Lock is a variable associated with data item which gives the status whether the possible operations can be applied on it or not.

A lock is a mechanism to control concurrent access to a data item.

Data items can be locked in two modes:

- **Exclusive(X) mode**. Data item can be both read as well as written. **X-lock** is requested using **lock-X** instruction.
- **Shared(S) mode**. Data item can only be read. **S-lock** is requested using **lock-S** instruction.

Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-Based Concurrency Control:

Locking ensures serializability by requiring that the access to data item be given in a mutually exclusive manner that is, while one transaction is accessing a data item, no other transaction can modify that data item. Thus, the intent of locking is to ensure serializability by ensuring mutual exclusion in accessing data items from the point of view of locking a database can be considered as being made up of a set of data items. A lock is a variable association with each such data item, manipulating the value of lock is called as locking. Locking is done by a subsystem of DBMS called LOCK MANAGER.

Exclusive Lock:

This mode of locking provides an exclusive use of data item to one particular transaction. The exclusive mode of locking is also called an UPDATE or a WRITE lock. If a transaction T locks a data item Q in an exclusive mode, no other transaction can access Q, not even to Read Q, until the lock is released by transaction T.

Shared Lock:

The shared lock is also called as a Read Lock. The intention of this mode of locking is to ensure that the data item does not undergo any modifications while it is locked in this mode. This mode allows several transactions to access the same item X if they all access X for reading purpose only. Thus, any number of transactions can concurrently lock and access a data item in the shared mode, but none of these transactions can modify the data item. A data item locked in the shared mode cannot be locked in the exclusive mode until the shared lock is released by all the transaction holding the lock. A data item locked in the exclusive mode cannot be locked in the shared mode until the exclusive lock on the data item is released.

IMPLEMENTING LOCK AND UNLOCK REQUESTS

- A transaction requests a shared lock on data item X by executing the Lock-S (X) instruction. Similarly, an exclusive lock is requested through the Lock-X (X) instruction. A data item X can be unlocked via the Unlock (X) instruction.
- When we use the shared / exclusive locking scheme, the lock manager should enforce the following rules:

  - A transaction T must issue a operation Lock-S (X) or Lock-X (X) before any Read (X) operation is performed in T.
  - The transaction T must issue the operation Lock-X (X) before any Write (X) operation is performed in T.
  - A transaction T must issue the operation Unlock (X) after all Read (X) and Write (X) operations are completed in T.
  - A transaction T will not issue a Lock-S (X) operation if it already holds an exclusive lock on item X.
  - A transaction T will not issue a Lock-X (X) operation if it already holds a shared or exclusive lock on item X.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item. But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

- The use of these two lock modes allows multiple transactions to read a data item but limits write access to just one transaction at a time.

Many locking protocols are available which indicate when a transaction may lock and unlock each of the data items.

Granting of Locks

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted.

Example of a transaction performing locking:

Let A and B be two accounts that are accessed by transactions T1 and T2. Transaction T1 transfers $50 from account B to account A .Transaction T2 displays the total amount of money in accounts A and B—that is, the sum A + B

*T*1:

lock-X(B);
read(B);
B := B − 50;
write(B);
unlock(B);
lock-X(A);
read(A);
A := A + 50;
write(A);
unlock(A).

*T*2:

lock-S(A);
read (A);
unlock(A);
lock-S(B);
read (B);
unlock(B);
display(A+B).

Suppose that the values of accounts A and B are $100 and $200, respectively. If these two transactions are executed serially, either in the order T1, T2 or the order T2, T1, then transaction T2 will display the value $300.

Note: Locking as above is ***not sufficient*** to guarantee serializability.

If, however, these transactions are executed concurrently, then schedule 1, is possible. In this case, transaction T2 displays $250, which is incorrect. The reason for this mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state.

The schedule shows the actions executed by the transactions, as well as the points at which the concurrency-control manager grants the locks. The transaction making a lock request cannot execute its next action until the concurrency control manager grants the lock. Hence, the lock must be granted in the interval of time between the lock-request operation and the following action of the transaction. Exactly when within this interval the lock is granted is not important; we can safely assume that the lock is granted just before the following action of the transaction.

Schedule 1

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Locking protocol

- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks, indicating when a transaction may lock and unlock each of the data items.
- Locking protocols enforce serializability by restricting the set of possible schedules.
- Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules

The Two-Phase Locking Protocol

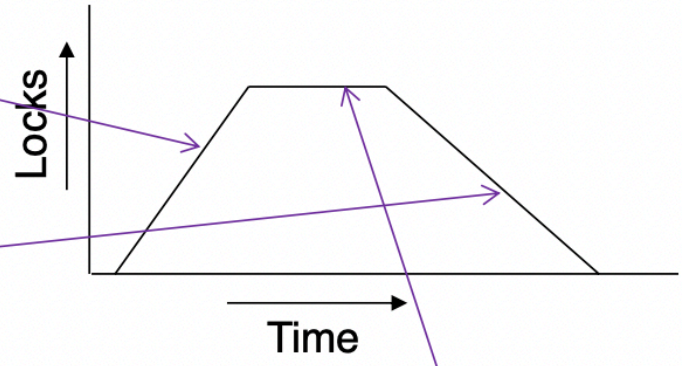
One protocol that ensures serializability is the two-phase locking (2PL) protocol. This protocol

requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase:** A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase**: A transaction may release locks, but may not obtain any new locks.

A transaction is said to follow Two Phase Locking Protocol if all locking operations precede the first unlock operation in a transaction. In other words release of locks on all data items required by the transaction have been acquired both the phases discussed earlier are monotonic. The number of locks are decreasing in the 2nd phase

- A protocol which ensures conflict-serializable schedules.
- Phase 1: **Growing Phase**
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: **Shrinking Phase**
  - Transaction may release locks
  - Transaction may not obtain locks
- The protocol assures serializability.
- It can be proved that the transactions can be serialized in the order of their **lock points** (i.e., the point where a transaction acquired its final lock).



Transaction T1 shown in Figure 1 below transfers $50 from account B to account A and transaction T2 in next Figure 2 displays the total amount of money in account A and B.

```
T1: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    unlock(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(A).
```

Figure 1: Transaction T1

```
T2: lock-S(A);
    read(A);
    unlock(A);
    lock-S(B);
    read(B);
    unlock(B);
    display(A + B).
```

Figure 2 : Transaction T2

Both the above transaction T1 and T2 do not follow Two Phase Locking Protocol. However transactions T3 and T4 (shown below) are in two phase.

T3:

Lock-X (B);

Read (B);

B := B – 50;

Write (B);

Lock-X (A);

Read (A);

A := A + 50;

Write (A);

Unlock (A);

Unlock (B);

T4:

Lock-S (A);

Read (A);

Lock-S (B);

Read (B);

Display (A + B);

Unlock (A);

Unlock (B);

Deadlock Handling

- A system is in a **deadlock** state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, \ldots, T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and . . . , and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.
- The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock. Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.
- Example:

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

  - Neither T_3 nor T_4 can make progress — executing **lock-S***(B)* causes T_4 to wait for T_3 to release its lock on B, while executing **lock-X***(A)* causes T_3 to wait for T_4 to release its lock on A.
  - Such a situation is called a **deadlock**.
    - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

There are **two principal methods for dealing with the deadlock** problem:

- ***Deadlock Prevention***
- ***Deadlock Detection and Recovery***

We can use a **deadlock prevention** protocol to ensure that the system will never enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection and deadlock recovery** scheme.

— Both methods may result in transaction rollback.
— Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient

Deadlock Prevention

- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies:

    1. Require that each transaction locks all its data items before it begins execution (pre-declaration).

    2. Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

The simplest scheme under the first approach requires that each transaction locks all its data items before it begins execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages to this protocol:

— it is often hard to predict, before the transaction begins, what data items need to be locked;
— data-item utilization may be very low, since many of the data items may be locked but unused for a long time.

Another approach for preventing deadlocks is to impose an ordering of all data items, and to require that a transaction lock data items only in a sequence consistent with the ordering

A variation of this approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a particular item, it cannot request locks on items that precede that item in the ordering.

The second approach for preventing deadlocks is to use **preemption** and transaction rollbacks. In preemption, when a transaction Tj requests a lock that transaction Ti holds, the lock granted to Ti may be preempted by rolling back of Ti , and granting of the lock to Tj . To control the preemption, we assign a unique timestamp, based on a counter or on the system clock, to each transaction when it begins. The system uses these timestamps only to decide whether a transaction should wait or roll back. Locking is still used for concurrency control. If a transaction is rolled back, it retains its old timestamp when restarted.

Two different deadlock-prevention schemes using timestamps:

1. The **wait–die** scheme is a nonpreemptive technique. When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp smaller than that of Tj (that is, Ti is older than Tj). Otherwise, Ti is rolled back (dies). For example, suppose that transactions T14, T15, and T16 have timestamps 5, 10, and 15, respectively. If T14 requests a data item held by T15, then T14 will wait. If T24 requests a data item held by T15, then T16 will be rolled back.
2. The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction Ti requests a data item currently held by Tj , Ti is allowed to wait only if it has a timestamp larger than that of Tj (that is, Ti is younger than Tj). Otherwise, Tj is rolled back (Tj is wounded by Ti).

The major problem with both of these schemes is that unnecessary rollbacks may occur.

Timeout-Based Schemes

Another simple approach to deadlock prevention is based on **lock timeouts**.

- In this approach, a transaction that has requested a lock waits for at most a specified amount of time. If the lock has not been granted within that time, the transaction is said to time out, and it rolls itself back and restarts.
- If there was in fact a deadlock, one or more transactions involved in the deadlock will time out and roll back, allowing the others to proceed.
- This scheme falls somewhere between deadlock prevention, where a deadlock will never occur, and deadlock detection and recovery.
- Ensures that deadlocks get resolved by timeout if they occur

- Simple to implement
- But may roll back transaction unnecessarily in absence of deadlock difficult to determine good value of the timeout interval.
- Starvation is also possible

Deadlock Detection and Recovery

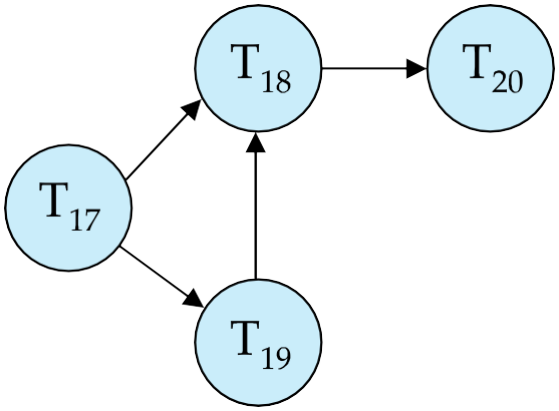
If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from deadlock. To do so, the system must:

- Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
- Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
- Recover from the deadlock when the detection algorithm determines that a deadlock exists.

Deadlock Detection

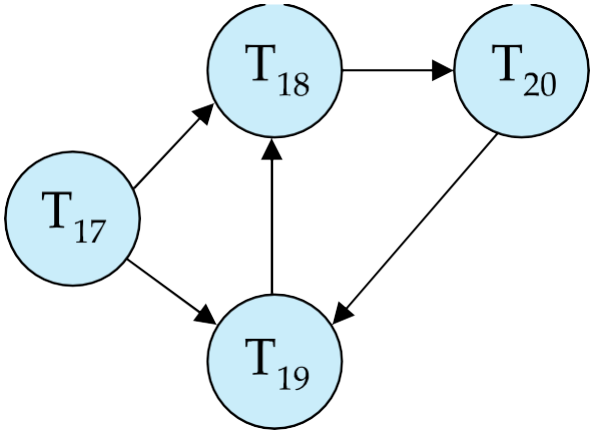
Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph.**

- **Wait-for graph**
  - *Vertices:* transactions
  - *Edge from* $T_i \rightarrow T_j$. : if T_i is waiting for a lock held in conflicting mode by T_j
- The system is in a deadlock state if and only if the wait-for graph has a cycle.
- Invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle

- Transaction T17 is waiting for transactions T18 and T19.
- Transaction T19 is waiting for transaction T18.
- Transaction T18 is waiting for transaction T20.



Wait-for graph with a cycle

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim**: Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock. We should roll back those transactions that will incur the minimum cost.
2. **Rollback:** Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
    - The simplest solution is a **total rollback**: Abort the transaction and then restart it.

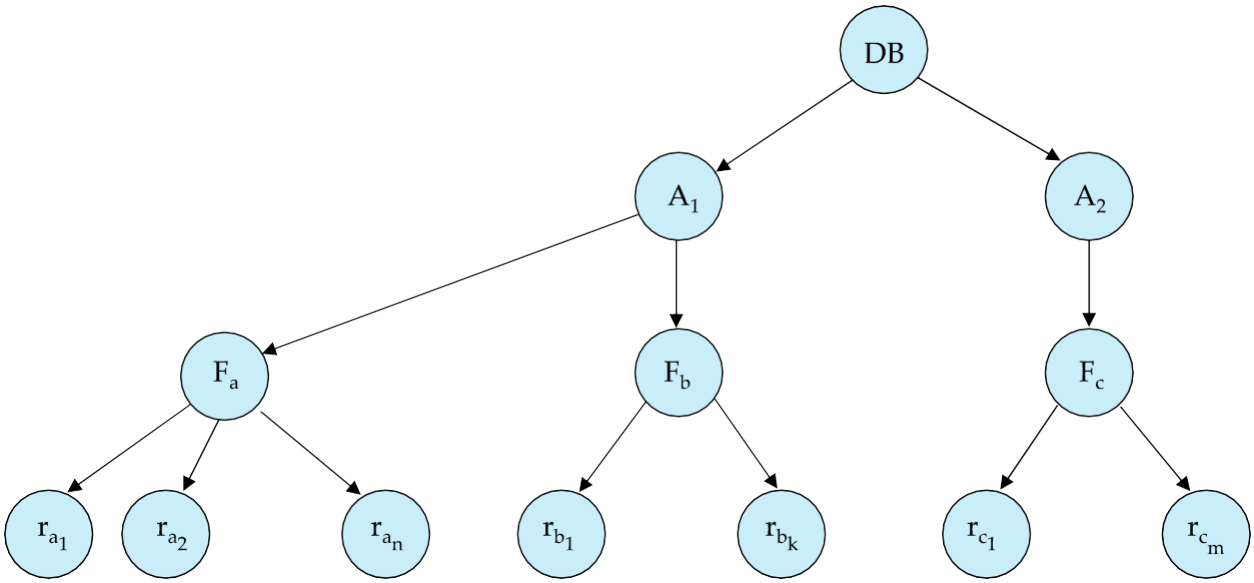
- **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for

3. **Starvation**: In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation.** We must ensure that a transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

Multiple Granularity

- Multiple granularity allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- Such a hierarchy can be represented graphically as a tree.
- A non-leaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode

Example of Granularity Hierarchy



The above tree consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No

file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
    - ***intention-shared*** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
    - ***intention-exclusive*** (IX): indicates explicit locking at a lower level with exclusive or shared locks
    - ***shared and intention-exclusive*** (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.
- intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes.

Compatibility Matrix with Intention Lock Modes

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules:
    1. The lock compatibility matrix must be observed.
    2. The root of the tree must be locked first, and may be locked in any mode.
    3. A node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
    4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
    5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
    6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.
- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- **Lock granularity escalation**: in case there are too many locks at a particular level, switch to higher granularity S or X lock

Timestamp-Based Protocols

The time stamping approach to scheduling concurrent transaction assigns a global unique time stamp to each transaction. The time stamp value produces an explicit order in which transactions are submitted to the DBMS. The time stamp must have following two properties:

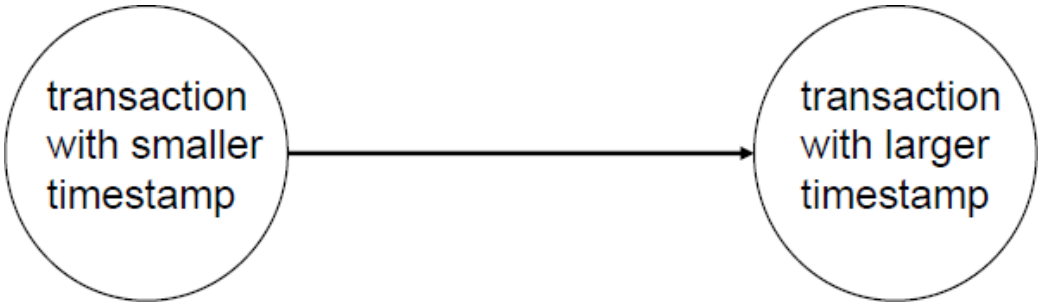
- **Uniqueness:** ensures that no equal time stamp values can exist.
- **Monotonicity:** ensures that time stamp values always increase.

All database operations (read and write) within the same transaction must have the same time stamp. The DBMS executes conflicting operations in time stamp order to ensure serializability of the transaction. If two transactions conflict, one often is stopped, rescheduling and assigned a new time stamps value.

Timestamps

- With each transaction Ti in the system, we associate a unique fixed timestamp, denoted by **TS(Ti).** This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction Ti has been assigned timestamp TS(Ti), and a new transaction Tj enters the system, then **TS(Ti) < TS(Tj).**
- There are two simple methods for implementing this scheme
  - Use the value of the **system clock** as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.
  - Use a **logical counter** that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.
- The timestamps of the transactions determine the serializability order.

  Thus, if **TS(Ti) < TS(Tj),** then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction **Ti appears before transaction Tj** .
- To implement this scheme, we associate with each data item Q two timestamp values:
  - **W-timestamp(Q):** denotes the largest timestamp of any transaction that executed write(Q) successfully.
  - **R-timestamp(Q)** : denotes the largest timestamp of any transaction that executed read(Q) successfully.
- These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.
- The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph.

The Timestamp-Ordering Protocol

The **timestamp-ordering protocol** ensures that any conflicting read and written operations are executed in timestamp order. This protocol operates as follows:

1. **Suppose that transaction Ti issues read(Q).**
   a. If **TS(Ti) < W-timestamp(Q)**, then Ti needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Ti is rolled back.
   b. If TS(Ti) ≥ W-timestamp(Q), then the read operation is executed, and **R-timestamp(Q)** is set to the maximum of **R-timestamp(Q)** and **TS(Ti).**
2. **Suppose that transaction Ti issues write(Q).**
   a. If **TS(Ti) < R-timestamp(Q),** then the value of Q that Ti is producing was needed previously, and the system assumed that that value would never be produced. Hence, the system rejects the write operation and rolls Ti back.
   b. If **TS(Ti) < W-timestamp(Q),** then Ti is attempting to write an obsolete value of Q. Hence, the system rejects this write operation and rolls Ti back.
   c. Otherwise, the system executes the write operation and sets **W-timestamp (Q)** to **TS(Ti).**

If a transaction Ti is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions T25 and T26. Transaction

T25 displays the contents of accounts A and B:

***T*25:**
read(*B*);
read(*A*);
display(*A* + *B*).

Transaction T26 transfers $50 from account B to account A, and then displays the contents of both:

***T*26:**
read(*B*);
***B* := *B* − 50;**
write(*B*);
read(*A*);
***A* := *A* + 50;**
write(*A*);
display(*A* + *B*).

In presenting schedules under the timestamp protocol, we shall assume that a transaction is assigned a timestamp immediately before its first instruction.

Thus, in following schedule**, TS(T25)< TS(T26)**, and the schedule is possible under the timestamp protocol

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)