Chapter 7: Concurrency Control

When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called *concurrency-control* schemes.

Lock-Based Protocols

One way to ensure serializability 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.

Locks

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. **Shared**: If a transaction *Ti* has obtained a **shared-mode lock** (denoted by S) on item *Q*, then *Ti* can read, but cannot write, *Q*.
2. **Exclusive**: If a transaction *Ti* has obtained an **exclusive-mode lock** (denoted by X) on item *Q*, then *Ti* can both read and write *Q*.

We require that every transaction **request** a lock in an appropriate mode on data item *Q*, depending on the types of operations that it will perform on *Q*. The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager **grants** the lock to the transaction.

To resolve the problem of concurrent requests, only compatible requests can be granted to proceed using the following Lock-compatibility matrix:

	S	X
S	true	false
X	false	false

Note that shared mode is compatible with shared mode, but not with exclusive mode. At any time, several shared-mode locks can be held simultaneously (by different transactions) on a particular data item. A subsequent exclusive-mode lock request has to wait until the currently held shared-mode locks are released.

A transaction requests a shared lock on data item *Q* by executing the lock-S(*Q*) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction. A transaction can unlock a data item *Q* by the unlock(*Q*) instruction.

To access a data item, transaction *Ti* must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will not grant the lock until all incompatible locks held by other transactions have been released. Thus, *Ti* is made to **wait** until all incompatible locks held by other transactions have been released.
Moreover, for a transaction to unlock a data item immediately after its final access of that data item is not always desirable, since serializability may not be ensured.

e.g.

1. *T1 transfers $50 B→ A where A = $100, B = $200*
2. *T2 displays A+B*

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_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

T2 displays $250; which is in correct

The reason of the mistake is that the transaction T1 unlocked data item B too early, as a result of which T2 saw an inconsistent state.

Deadlock

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)	

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule described above for *T*3 and *T*4. Since *T*3 is holding an exclusive-mode lock on *B* and *T*4 is requesting a shared-mode lock on *B*, *T*4 is waiting for *T*3 to unlock *B*. Similarly, since *T*4 is holding a shared-mode lock on *A* and *T*3 is requesting an exclusive-mode lock on *A*, *T*3 is waiting for *T*4 to unlock *A*. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called **deadlock**.

When deadlock occurs, the system must roll back one of the two transactions. Once a transaction has been rolled back, the data items that were locked by that transaction are unlocked. These data items are then available to the other transaction, which can continue with its execution.

Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back of transactions, where as inconsistent states may lead to real-world problems that cannot be handled by the database system. We shall require that each transaction in the system follow a set of rules, called a **locking protocol**, indicating when a transaction may lock and unlock each of the data items. Locking protocols restrict the number of possible schedules. The set of all such schedules is a proper subset of all possible serializable schedules.

We say that a schedule *S* is **legal** under a given locking protocol if *S* is a possible schedule for a set of transactions that follow the rules of the locking protocol. We say that a locking protocol **ensures** conflict serializability if and only if all legal schedules are conflict serializable

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. However, care must be taken to avoid the following scenario.

Suppose a transaction *T*2 has a shared-mode lock on a data item, and another transaction *T*1 requests an exclusive-mode lock on the data item. Clearly, *T*1 has to wait for *T*2 to release the shared-mode lock. Meanwhile, a transaction *T*3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to *T*2, so *T*3 may be granted the shared-mode lock. At this point *T*2 may release the lock, but still *T*1 has to wait for *T*3 to finish. But again, there may be a new transaction *T*4 that requests a shared-mode lock on the same data item, and is granted the lock before *T*3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but *T*1 never gets the exclusive-mode lock on the data item. The transaction *T*1 may never make progress, and is said to be ***starved.***

We can avoid starvation of transactions by granting locks in the following manner: When a transaction *Ti* requests a lock on a data item *Q* in a particular mode *M*, the concurrency-control manager grants the lock provided that

i. There is no other transaction holding a lock on *Q* in a mode that conflicts with *M*.
ii. There is no other transaction that is waiting for a lock on *Q*, and that made its lock request before *Ti*.

Thus, a lock request will never get blocked by a lock request that is made later.

The Two-Phase Locking Protocol

One protocol that ensures serializability is the **two-phase locking 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.

We can show that the two-phase locking protocol ensures conflict serializability. Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions.

Two-phase locking does not ensure freedom from deadlock. Cascading rollback may occur under two-phase locking.

Cascading rollbacks can be avoided by a modification of two-phase locking called the ***strict two-phase locking protocol***. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

Another variant of two-phase locking is the ***rigorous two-phase locking protocol***, which requires that all locks be held until the transaction commits. We can easily verify that, with rigorous two-phase locking, transactions can be serialized in the order in which they commit. Most database systems implement either strict or rigorous two-phase locking.
Consider the following two transactions, for which we have shown only some of the significant read and write operations:

*T*8: **read(*a*1);**
read(*a*2);
. . .
read(*an*);
write(*a*1).

*T*9: **read(*a*1);**
read(*a*2);
display(*a*1 + *a*2).

If we employ the two-phase locking protocol, then *T*8 must lock *a*1 in exclusive mode. Therefore, any concurrent execution of both transactions amounts to a serial execution. Notice, however, that *T*8 needs an exclusive lock on *a*1 only at the end of its execution, when it writes *a*1. Thus, if *T*8 could initially lock *a*1 in shared mode, and then could later change the lock to exclusive mode, we could get more concurrency, since *T*8 and *T*9 could access *a*1 and *a*2 simultaneously.

This observation leads us to a refinement of the basic two-phase locking protocol, in which ***lock conversions*** are allowed. We shall provide a mechanism for upgrading a shared lock to an exclusive lock, and downgrading an exclusive lock to a shared lock. We denote conversion from shared to exclusive modes by **upgrade**, and from exclusive to shared by **downgrade**. Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.

Returning to our example, transactions *T*8 and *T*9 can run concurrently under the refined two-phase locking protocol, as shown in the incomplete schedule describe below, where only some of the locking instructions are shown.

T_8	T_9
lock-S(a_1)	
	lock-S(a_1)
lock-S(a_2)	
	lock-S(a_2)
lock-S(a_3)	
lock-S(a_4)	
	unlock(a_1)
	unlock(a_2)
lock-S(a_n)	
upgrade(a_1)	

Note that a transaction attempting to upgrade a lock on an item *Q* may be forced to wait. This enforced wait occurs if *Q* is currently locked by *another* transaction in shared mode.

Just like the basic two-phase locking protocol, two-phase locking with lock conversion generates only conflict-serializable schedules, and transactions can be serialized by their lock points. Further, if exclusive locks are held until the end of the transaction, the schedules are cascadeless.

Graph-Based Protocols

The two-phase locking protocol is both necessary and sufficient for ensuring serializability in the absence of information concerning the manner in which data items are accessed. But, if we wish to develop protocols that are not two phase, we need additional information on how each transaction will access the database. There are various models that can give us the additional information, each differing in the amount of information provided. The simplest model requires that we have prior knowledge about the order in which the database items will be accessed. Given such information, it is possible to construct locking protocols that are not two phase, but that, nevertheless, ensure conflict serializability.

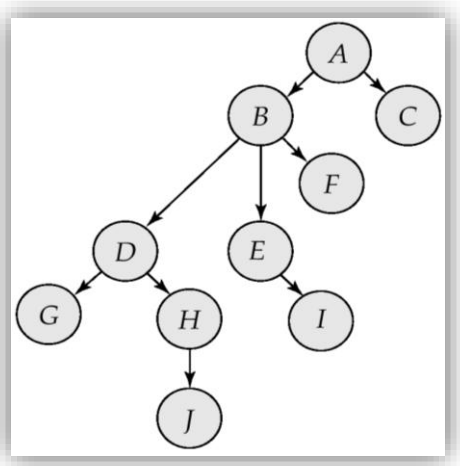
To acquire such prior knowledge, we impose a partial ordering → on the set **D** = *{d*1, *d*2, . . ., *dh}* of all data items. If *di* → *dj* , then any transaction accessing both *di* and *dj* must access *di* before accessing *dj* . This partial ordering may be the result of either the logical or the physical organization of the data, or it may be imposed solely for the purpose of concurrency control.

The partial ordering implies that the set**D**may now be viewed as a directed acyclic graph, called a **database graph**. In this section, for the sake of simplicity, we will restrict our attention to only those graphs that are rooted trees. We will present a simple protocol, called the *tree protocol*, which is restricted to employ only *exclusive* locks.

In the **tree protocol**, the only lock instruction allowed is lock-X. Each transaction *Ti* can lock a data item at most once, and must observe the following rules:

1. The first lock by *Ti* may be on any data item.
2. Subsequently, a data item *Q* can be locked by *Ti* only if the parent of *Q* is currently locked by *Ti*.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by *Ti* cannot subsequently be relocked by *Ti*.

All schedules that are legal under the tree protocol are conflict serializable. To illustrate this protocol, consider the database graph described below.



The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

T10: lock-X(*B*); lock-X(*E*); lock-X(*D*); unlock(*B*); unlock(*E*); lock-X(*G*); unlock(*D*); unlock(*G*).

T11: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).

T12: lock-X(*B*); lock-X(*E*); unlock(*E*); unlock(*B*).

T13: lock-X(*D*); lock-X(*H*); unlock(*D*); unlock(*H*).

One possible schedule in which these four transactions participated appears in Figure below.

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock. The tree protocol does not ensure recoverability and cascadelessness. To ensure recoverability and cascadelessness, the protocol can be modified to not permit release of exclusive locks until the end of the transaction. Holding exclusive locks until the end of the transaction reduces concurrency.

The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times and to an increase in concurrency.

However, the protocol has the disadvantage that, in some cases, a transaction may have to lock data items that it does not access. For example, a transaction that needs to access data items *A* and *J* in the database graph must lock not only *A* and *J*, but also data items *B*, *D*, and *H*. This additional locking result in increased locking overhead, the possibility of additional waiting time, and a potential decrease in concurrency.

Timestamp-Based Protocols

The locking protocols that we have described thus far determine the order between every pair of conflicting transactions at execution time by the first lock that both members of the pair request that involves incompatible modes. Another method for determining the serializability order is to select an ordering among transactions in advance. The most common method for doing so is to use a *timestamp-ordering* scheme.

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:

1. 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.
2. 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:

a. **W-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed write(*Q*) successfully.

b. **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

The **timestamp-ordering protocol** ensures that any conflicting read and write 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 writes operation, the system assigns it a new timestamp and restarts it.

To illustrate this protocol, we consider transactions *T*14 and *T*15. Transaction *T*14 displays the contents of accounts *A* and *B*:

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

We note that the preceding execution can also be produced by the two-phase locking protocol. There are, however, schedules that are possible under the two-phase locking protocol, but are not possible under the timestamp protocol, and vice versa.

The timestamp-ordering protocol ensures conflict serializability. This is because conflicting operations are processed in timestamp order. The protocol ensures freedom from deadlock, since no transaction ever waits. However, there is a possibility of starvation of long transactions if a sequence of conflicting short transactions causes repeated restarting of the long transaction. If a transaction is found to be getting restarted repeatedly, conflicting transactions need to be temporarily blocked to enable the transaction to finish. The protocol can generate schedules that are not recoverable. However, it can be extended to make the schedules recoverable, in one of several ways:

- Recoverability and cascadelessness can be ensured by performing all writes together at the end of the transaction. The writes must be atomic in the following sense: While the writes are in progress, no transaction is permitted to access any of the data items that have been written.
- Recoverability and cascadelessness can also be guaranteed by using a limited form of locking, whereby reads of uncommitted items are postponed until the transaction that updated the item commits.

Validation-Based Protocols

In cases where a majority of transactions are read-only transactions, the rate of conflicts among transactions may be low. Thus, many of these transactions, if executed without the supervision of a concurrency-control scheme, would nevertheless leave the system in a consistent state. A concurrency-control scheme imposes overhead of code execution and possible delay of transactions. It may be better to use an alternative scheme that imposes less overhead. A difficulty in reducing the overhead is that we do not know in advance which transactions will be involved in a conflict. To gain that knowledge, we need a scheme for **monitoring** the system.

We assume that each transaction *Ti* executes in two or three different phases in its lifetime, depending on whether it is a read-only or an update transaction. The phases are, in order,

1. **Read phase**. During this phase, the system executes transaction *Ti*. It reads the values of the various data items and stores them in variables local to *Ti*. It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase**. Transaction *Ti* performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.
3. **Write phase**. If transaction *Ti* succeeds in validation (step 2), then the system applies the actual updates to the database. Otherwise, the system rolls back *Ti*.

Each transaction must go through the three phases in the order shown. However, all three phases of concurrently executing transactions can be interleaved.

To perform the validation test, we need to know when the various phases of transactions *Ti* took place. We shall, therefore, associate three different timestamps with transaction *Ti*:

1. **Start**(*Ti*), the time when *Ti* started its execution.
2. **Validation**(*Ti*), the time when *Ti* finished its read phase and started its validation phase.
3. **Finish**(*Ti*), the time when *Ti* finished its write phase.

We determine the serializability order by the timestamp-ordering technique, using the value of the timestamp Validation(*Ti*). Thus, the value TS(*Ti*) = Validation(*Ti*) and, if TS(*Tj*) < TS(*Tk*), then any produced schedule must be equivalent to a serial schedule in which transaction *Tj* appears before transaction *Tk*. The reason we have chosen Validation(*Ti*), rather than Start(*Ti*), as the timestamp of transaction *Ti* is that we can expect faster response time provided that conflict rates among transactions are indeed low.

The **validation test** for transaction *Tj* requires that, for all transactions *Ti* with TS(*Ti*) < TS(*Tj*), one of the following two conditions must hold:

- Finish(*Ti*) < Start(*Tj*). Since *Ti* completes its execution before *Tj* started, the serializability order is indeed maintained.
- The set of data items written by *Ti* does not intersect with the set of data items read by *Tj*, and *Ti* completes its write phase before *Tj* starts its validation phase (Start(*Tj*) < Finish(*Ti*) < Validation(*Tj*)). This condition ensures that the writes of *Ti* and *Tj* do not overlap. Since the writes of *Ti* do not affect the read of *Tj* , and since *Tj* cannot affect the read of *Ti*, the serializability order is indeed maintained.

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	read(A)
	$A := A + 50$
read(A)	
⟨*validate*⟩	
display($A + B$)	
	⟨*validate*⟩
	write(B)
	write(A)

The validation scheme automatically guards against cascading rollbacks, since the actual writes take place only after the transaction issuing the write has committed. However, there is a possibility of starvation of long transactions, due to a sequence of conflicting short transactions that cause repeated restarts of the long transaction. To avoid starvation, conflicting transactions must be temporarily blocked, to enable the long transaction to finish.

This validation scheme is called the **optimistic concurrency control** scheme since transactions execute optimistically, assuming they will be able to finish execution and validate at the end. In contrast, locking and timestamp ordering are pessimistic in that they force a wait or a rollback whenever a conflict is detected, even though there is a chance that the schedule may be conflict serializable.

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. None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem. 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.

Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.

Deadlock Prevention

There are two approaches to deadlock prevention. One approach ensures that no cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together. The other approach is closer to deadlock recovery, and performs transaction rollback instead of waiting for a lock, whenever the wait could potentially result in a deadlock.

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: (1) it is often hard to predict, before the transaction begins, what data items need to be locked; (2) 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. We have seen one such scheme in the tree protocol, which uses a partial ordering of data items. 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. This scheme is easy to implement, as long as the set of data items accessed by a transaction is known when the transaction starts execution.

The second approach for preventing deadlocks is to use preemption and transaction rollbacks. In preemption, when a transaction *T*2 requests a lock that transaction *T*1 holds, the lock granted to *T*1 may be **preempted** by rolling back of *T*1, and granting of the lock to *T*2. To control the preemption, we assign a unique timestamp to each transaction. 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 have been proposed:

[1] The ***wait–die*** scheme is a non preemptive 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 *T*22, *T*23, and *T*24 have timestamps 5, 10, and 15, respectively. If *T*22 requests a data item held by *T*23, then *T*22 will wait. If *T*24 requests a data item held by *T*23, then *T*24 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*). Returning to our example, with transactions *T*22, *T*23, and *T*24, if *T*22 requests a data item held by *T*23, then the data item will be preempted from *T*23, and *T*23 will be rolled back. If *T*24 requests a data item held by *T*23, then *T*24 will wait.

Whenever the system rolls back transactions, it is important to ensure that there is no **starvation**—that is, no transaction gets rolled back repeatedly and is never allowed to make progress.

Both the wound–wait and the wait–die schemes avoid starvation: At any time, there is a transaction with the smallest timestamp. This transaction *cannot* be required to roll back in either scheme. Since timestamps always increase, and since transactions are *not* assigned new timestamps when they are rolled back, a transaction that is rolled back repeatedly will eventually have the smallest timestamp, at which point it will not be rolled back again.

There are, however, significant differences in the way that the two schemes operate.

i. In the wait–die scheme, an older transaction must wait for a younger one to release its data item. Thus, the older the transaction gets, the more it tends to wait. By contrast, in the wound–wait scheme, an older transaction never waits for a younger transaction.

ii. In the wait–die scheme, if a transaction Ti dies and is rolled back because it requested a data item held by transaction Tj, then Ti may reissue the same sequence of requests when it is restarted. If the data item is still held by Tj , then Ti will die again. Thus, Ti may die several times before acquiring the needed data item. Contrast this series of events with what happens in the wound–wait scheme. Transaction Ti is wounded and rolled back because Tj requested a data item that it holds. When Ti is restarted and requests the data item now being held by Tj , Ti waits. Thus, there may be fewer rollbacks in the wound–wait scheme.

Timeout-Based Schemes

Another simple approach to deadlock handling 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.

It is hard to decide how long a transaction must wait before timing out. Too long a wait results in unnecessary delays once a deadlock has occurred. Too short a wait results in transaction rollback even when there is no deadlock, leading to wasted resources. Starvation is also a possibility with this scheme. Hence, the timeout-based scheme has limited applicability.

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 the deadlock. To do so, the system must:

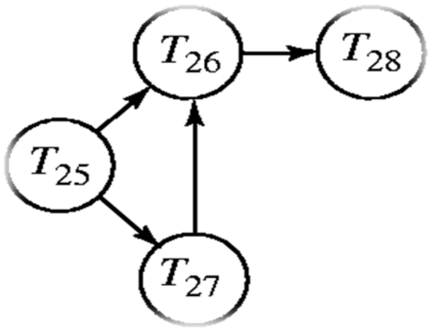
1. Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
2. Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
3. Recover from the deadlock when the detection algorithm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

Deadlock Detection

Deadlocks can be described precisely in terms of a directed graph called a **wait-for graph**. 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 in the system. Each element in the set E of edges is an ordered pair $Ti \rightarrow Tj$. If $Ti \rightarrow Tj$ is in E, then there is a directed edge from transaction *Ti* to *Tj* , implying that transaction *Ti* is waiting for transaction *Tj* to release a data item that it needs. When transaction *Ti* requests a data item currently being held by transaction *Tj* , then the edge *Ti* → *Tj* is inserted in the wait-for graph. This edge is removed only when transaction *Tj* is no longer holding a data item needed by transaction *Ti*.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

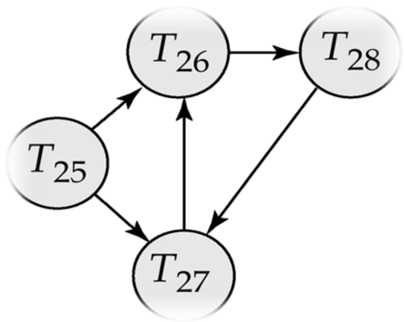
To illustrate these concepts, consider the wait-for graph in Figure below, which depicts the following situation:



Transaction *T*25 is waiting for transactions *T*26 and *T*27.
Transaction *T*27 is waiting for transaction *T*26.
Transaction *T*26 is waiting for transaction *T*28.

Since the graph has no cycle, the system is not in a deadlock state.

Suppose now that transaction *T*28 is requesting an item held by *T*27. The edge *T*28 → *T*27 is added to the wait-for graph, resulting in the new system state in Figure. This time, the graph contains the cycle



*T*26 → *T*28 →*T*27 →*T*26 implying that transactions *T*26, *T*27, and *T*28 are all deadlocked.

Consequently, the question arises: When should we invoke the detection algorithm?
The answer depends on two factors:

- How often does a deadlock occur?
- How many transactions will be affected by the deadlock?

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. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including

- How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
- How many data items the transaction has used.
- How many more data items the transaction needs for it to complete.
- How many transactions will be involved in the rollback.

[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. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions.

Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback.

[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 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.