Figure 12.5 Two transactions; one modified while the other reads.

Transaction T_5	Transaction T_6
Read(A)	$Sum := 0$
$A := A - 100$	**Read**(A)
Write(A)	$Sum := Sum + A$
Read(B)	**Read**(B)
$B := B + 100$	$Sum := Sum + B$
Write(B)	**Write**(Sum)

Consider the transactions of Figure 12.5. Suppose A and B represent some data-items containing integer valued data, for example, two accounts in a bank (or a quantity of some part X in two different locations, etc.). Let us assume that transaction T_5 transfers 100 units from A to B. Transaction T_6 is concurrently running and it wants to find the total of the current values of data-items A and B (the sum of the balance in case A and B represent two accounts, or the total quantity of part X in the two different locations, etc.).

Figure 12.6 gives a possible schedule for the concurrent execution of the transactions of Figure 12.5 with the initial value of A and B being 500 and 1000, respectively. We notice from the schedule that transaction T_6 uses the value of A before the transfer was made, but it uses the modified value of B after the transfer. The result is that transaction T_6 erroneously determines the total of A and B as being 1600 instead of 1500. We can also come up with another schedule of the concurrent exe-

Figure 12.6 Example of inconsistent reads.

	Schedule	Transaction T_5	Transaction T_6	Value of Database items		
				A	B	Sum
	Read(A)	**Read**(A)		500	1100	—
	$Sum := 0$		Sum := 0			0
T	**Read** (A)		**Read** (A)			
i	$A := A - 100$	$A := A - 100$				
m	**Write**(A)	**Write**(A)		400		
e	$Sum := Sum + A$		$Sum := Sum + A$			500
	Read(B)	**Read**(B)				
	$B :=$ B + 100	$B := B + 100$				
	Write(B)	**Write**(B)			1100	
	Read(B)		**Read**(B)			
	$Sum := Sum + B$		$Sum := Sum + B$			
↓	**Write**(Sum)		**Write**(Sum)			1600

In effect, the division of a transaction into interdependent transactions run serially in the wrong order would give erroneous results. Furthermore, these interdependent transactions must not be run concurrently, otherwise the concurrent execution will lead to results that could be incorrect again and not agree with the result obtained by any serial execution of the same transactions. It is a logical error to divide a single set of operations into two or more transactions. We assume hereafter that transactions are semantically correct.

12.2 Serializability

Let us reconsider the transactions ot Figure 12.3. We assume that these transactions are independent. An execution schedule of these transactions as shown in Figure 12.7 is called a **serial execution.** In a serial execution, each transaction runs to completion before any statements from any other transaction are executed. In Schedule A given in Figure 12.7a, transaction T_3 is run to completion before transaction T_4 is executed. In Schedule B, transaction T_4 is run to completion before transaction T_3 is started. If the initial value of *A* in the database were 200, Schedule A would result in the value of A being changed to 231. Similarly, Schedule B with the same initial value of *A* would give a result of 230.

This may seem odd, but in a shared environment, the result obtained by independent transactions that modify the same data-item always depends on the order in which these transactions are run; and any of these results is considered to be correct.

Figure 12.7 Two serial schedules.

	Schedule A	Transaction T_3	Transaction T_4
	Read(*A*)	**Read**(*A*)	
r	$A := A + 10$	$A := A + 10$	
i	**Write**(*A*)	**Write**(*A*)	
m	**Read**(*A*)		**Read**(*A*)
e	$A := A * 1.1$		$A := A * 1.1$
↓	**Write**(*A*)		**Write**(*A*)

(a)

	Schedule B	Transaction T_3	Transaction T_4
	Read(*A*)		**Read**(*A*)
T	$A := A * 1.1$		$A := A * 1.1$
i	**Write**(*A*)		**Write**(*A*)
m	**Read**(*A*)	**Read**(*A*)	
e	$A := A + 10$	$A := A + 10$	
↓	**Write**(*A*)	**Write**(*A*)	

(b)

If there are two transactions and if they refer to and use distinct data-items, the result obtained by the interleaved execution of the statements of these transactions would be the same regardless of the order in which these statements are executed (provided there are no other concurrent transactions that refer to any of these data-items). In this chapter, we assume that the concurrent transactions share some data-items, hence we are interested in a correct ordering of execution of the statements of these transactions.

A nonserial schedule wherein the operations from a set of concurrent transactions are interleaved is considered to be serializable if the execution of the operations in the schedule leaves the database in the same state as some serial execution of these transactions. With two transactions, we can have at most two distinct serial schedules, and starting with the same state of the database, each of these serial schedules could give a different final state of the database. Starting with an initial value of 200 for *A*, the serial schedule illustrated in Figure 12.7a would give the final value of *A* as 231, and for the serial schedule illustrated in part b the final value of *A* would be 230. If we have n concurrent transactions, it is possible to have n!, where n! = n * (n − 1) * (n − 2) * . . . * 3 * 2 * 1 distinct serial schedules, and possibly that many distinct resulting modifications to the database. For a serializable schedule, all we require is that the schedule gives a result that is the same as any one of these possibly distinct results.

When n transactions are run concurrently and in an interleaved manner, the number of possible schedules is much larger than n!. We would like to find out if a given interleaved schedule produces the same result as one of the serial schedules. If the answer is positive, then the given interleaved schedule is said to be serializable.

Definition: **Serializable Schedule:**

Given an interleaved execution of a set of n transactions; the following conditions hold for each transaction in the set:

- All transactions are correct in the sense that if any one of the transactions is executed by itself on a consistent database, the resulting database will be consistent.
- Any serial execution of the transactions is also correct and preserves the consistency of the database; the results obtained are correct. (This implies that the transactions are logically correct and that no two transactions are interdependent).

The given interleaved execution of these transactions is said to be serializable if it produces the same result as some serial execution of the transactions.

Since a serializable schedule gives the same result as some serial schedule and since that serial schedule is correct, then the serializable schedule is also correct. Thus, given any schedule, we can say it is correct if we can show that it is serializable.

Algorithm 12.1 given in Section 12.2.2 establishes the serializability of an arbitrarily interleaved execution of a set of transactions on a database. The algorithm does not consider the nature of the computations performed by a transaction nor the transactions T_{12} and T_{13}. In the precedence graph there is an edge from T_{12} to T_{13} as well as an edge from T_{13} to T_{12}. The edge T_{13} to T_{12} is included because T_{12} executes a write operation after T_{13} executes a write operation for the same database item *A*. The edge T_{12} to T_{13} is included because T_{13} executes a write operation after T_{12} executes a read operation for the same database item *A*. We see that the precedence graph has a cycle, since we can start from one of the nodes of the graph and, following the directed edges, return to the starting node.

A precedence graph is said to be **acyclic** if there are no cycles in the graph. The graph of Figure 12.8b has no cycles. The graph of Figure 12.9b is **cyclic,** since it has a cycle.

The precedence graph for serializable schedule S must be acyclic, hence it can be converted to a serial schedule. To test for the serializability of the arbitrary schedule S for transactions $T_1, \ldots, T_k$ we convert the schedule into a precedence graph and then test the precedence graph for cycles. If no cycles are detected, the schedule is serializable; otherwise it is not. If there are n nodes in the graph for schedule S, the number of operations required to check if there is a cycle in the graph is proportional to n^2.

Algorithm 12.1

Serializability Test

Input: Schedule S for the concurrent execution of transactions $T_1, \ldots, T_k$.

Output: A serial schedule for S if one exists.

Step 1: Create precedence graph G as follows. Transactions $T_1, \ldots, T_k$ are the nodes and each edge of the graph is inserted as follows: For a database item X used in the schedule find an operation Write(X) for some transaction T_i; if there is a subsequent (earliest) operation Read(X) in transaction T_j, $i \neq j$, insert an edge from T_i to T_j in the precedence graph, since T_i must be executed before T_j. For a database item X, if T_j executes a Write(X) after T_i, $i \neq j$, executes a Read(X) or a Write(X) operation, insert an edge from T_i to T_j in the precedence graph.

Step 2: If the graph G has a cycle (see Exercise 12.6), schedule S is nonserializable. If G is acyclic, then find, using the topological sort given below, a linear ordering of the transactions so that if there is an arc from T_i to T_j in G, T_i precedes T_j. Find a serial schedule as follows:

(a) Initialize the serial schedule as empty.

(b) Find a transaction T_i [illegible] T_i is the next transaction in the serial schedule.

(c) Remove T_i and all edges emitting from T_i. If the [illegible] nonempty), return to (b), otherwise the serial schedule is complete.

12.2.2 Serializability Algorithm: Read-before-Write Protocol

In the **read-before-write protocol** we assume that a transaction will read the data-item before it modifies it and after modifications, the modified value is written back to the database. In Algorithm 12.1, we give the method of testing whether a schedule is serializable. We create a precedence graph and test for a cycle in the graph. If we find a cycle, the schedule is nonserializable; otherwise we find a linear ordering of the transactions.

In Examples 12.1 and 12.2 we illustrate the application of this algorithm.

Example 12.1

Consider the schedule of Figure A. The precedence graph for this schedule is given in Figure B. The graph has three nodes corresponding to the three transactions T_{14}, T_{15}, and T_{16}. There is an arc from T_{14} to T_{15} because T_{14} writes data-item A before T_{15} reads it. Similarly, there is an arc from T_{15} to T_{16} because T_{15} writes data-item B before T_{16} reads it. Finally, there is an arc from T_{16} to T_{14} because T_{16} writes data-item C before T_{14} reads it. The precedence graph of Figure B has a cycle formed by the directed edges from T_{14} to T_{15}, from T_{15} to T_{16} and from T_{16} back to T_{14}. Hence, the schedule of Figure A is not serializable. We cannot execute the three transactions serially to get the same result as the given schedule.

Figure A An execution schedule involving three transactions.

	Schedule	Transaction T_{14}	Transaction T_{15}	Transaction T_{16}
	Read(A)	**Read**(A)		
	Read(B)		**Read**(B)	
	$A := f_1(A)$	$A := f_1(A)$		
	Read(C)			**Read**(C)
	$B := f_2(B)$		$B := f_2(B)$	
T	**Write**(B)		**Write**(B)	
i	$C := f_3(C)$			$C := f_3(C)$
m	**Write**(C)			**Write**(C)
e	**Write**(A)	**Write**(A)		
	Read(B)			**Read**(B)
	Read(A)		**Read**(A)	
	$A := f_4(A)$		$A := f_4(A)$	
	Read(C)	**Read**(C)		
	Write(A)		**Write**(A)	
	$C := f_5(C)$	$C := f_5(C)$		
	Write(C)	**Write**(C)		
	$B := f_6(B)$			$B := f_6(B)$
	Write(B)			**Write**(B)

changed between the time of reading and of writing. In the multiversion technique, a data-item is never written over; each write operation creates a new version of a data-item. Many versions of a data-item exist and these represent the historical evolution of the data-item. A transaction sees the data-item of its own epoch. Conflicts are resolved by rollback of a transaction that is too late to write out all values from its epoch. We examine each of these concurrency control schemes in the following sections. The problem of deadlock, which is possible in some of these schemes and/ or their modifications, is discussed in Section 12.8.

12.4 Locking Scheme

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 associated with each such data-item. Manipulating the value of a lock is called **locking.** The value of a lock variable is used in the locking scheme to control the concurrent access and manipulation of the associated data-item. Locking the items being used by a transaction can prevent other concurrently running transactions from using these locked items. The locking is done by a subsystem of the database management system usually called the **lock manager.**

So that concurrency is not restricted unnecessarily, at least two types of locks are defined: exclusive lock and shared lock.

Exclusive lock: The exclusive lock is also called an update or a write lock. The intention of this mode of locking is to provide exclusive use of the data-item to one transaction. 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 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. 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 a shared mode cannot be locked in the exclusive mode until the shared lock is released by all transactions 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.

The protocol of sharing is as follows. Each transaction, before accessing a data-item, requests that the data-item be locked in the appropriate mode. If the data-item is not locked, the lock request is honored by the lock manager. If the data-item is already locked, the request may or may not be granted, depending on the mode of locking requested and the current mode in which the data-item is locked. If the mode of locking requested is shared and if the data-item is already locked in the shared mode, the lock request can be granted. If the data-item is locked in an exclusive mode, then the lock request cannot be granted, regardless of the mode of the request. In this case the requesting transaction has to wait till the lock is released.

The compatibility of a lock request for a data-item with respect to its current state of locking is given in Figure 12.10. Here we are assuming that the request for locking is made by a transaction not already holding a lock on the data-item.

If transaction T_x makes a request to lock data item A in the shared mode and if A is not locked or if it is already locked in the shared mode, the lock request is granted. This means that a subsequent request from another transaction, T_y, to lock

Figure 12.10 Compatibility of locking.

Lock mode of request	Current state of locking of data-item Unlocked	Shared	Exclusive
Unlock		yes	yes
Shared	yes	yes	no
Exclusive	yes	no	no

data-item *A* in the exclusive mode would not be granted and transaction T_y will have to wait until *A* is unlocked. While *A* is locked in the shared mode, if transaction T_z makes a request to lock it in the shared mode, this request can be granted. Both T_x and T_z can concurrently use data-item *A*.

If transaction T_x makes a request to lock data-item *A* in the shared mode and if *A* is locked in the exclusive mode, the request made by transaction T_x cannot be granted. Similarly, a request by transaction T_z to lock *A* in the exclusive mode while it is already locked in the exclusive mode would also result in the request not being granted, and T_z would have to wait until the lock on *A* is released.

From the above we see that any lock request for a data-item can only be granted if it is compatible with the current mode of locking of the data-item. If the request is not compatible, the requesting transaction has to wait until the mode becomes compatible.

The releasing of a lock on a data-item changes its lock status. If the data-item was locked in an exclusive mode, the release of lock request by the transaction holding the exclusive lock on the data-item would result in the data-item being unlocked. Any transaction waiting for a release of the exclusive lock would have a chance of being granted its request for locking the data-item. If more than one transaction is waiting, it is assumed that the lock manager would use some fair scheduling technique to choose one of these waiting transacuons.

If the data-item was locked in a shared mode, the release of lock request by the transaction holding the shared lock on the data-item may not result in the data-item being unlocked. This is because more than one transaction may be holding a shared lock on the data-item. Only when the transaction releasing the lock is the only transaction having the shared lock does the data-item become unlocked. The lock manager may keep a count of the number of transactions holding a shared lock on a data-item. It would increase this value by one when an additional transaction is granted a shared lock and decrease the value by one when a transaction holding a shared lock releases the lock. The data-item would then become unlocked when the number of transactions holding a shared lock on it becomes zero. This count could be stored in an appropriate data structure along with the data-item but it would be accessible only to the lock manager.

The lock manager must have a priority scheme whereby it decides whether to allow additional transactions to lock a data-item in the share-mode in the following situation:

- The data-item is already locked in the shared mode.
- There is at least one transaction waiting to lock the data-item in the exclusive mode.

Figure 12.14 A possible solution to the inconsistent read problem.

Time	Schedule	Transaction T_{20}	Transaction T_{21}
	Lockx(*Sum*)		**Lockx**(*Sum*)
	Sum := 0		*Sum* := 0
	Locks(*A*)		**Locks**(*A*)
	Read(*A*)		**Read**(*A*)
	Sum := *Sum* + *A*		*Sum* := *Sum* + *A*
T	**Locks**(*B*)		**Locks**(*B*)
i	**Read**(*B*)		**Read**(*B*)
m	*Sum* := *Sum* + *B*		*Sum* := Sum + B
e	**Write**(*Sum*)		**Write**(*Sum*)
	Unlock(*B*)		**Unlock**(*B*)
	Unlock(*A*)		**Unlock**(*A*)
	Unlock(*Sum*)		**Unlock**(*Sum*)
	Lockx(*A*)	**Lockx**(*A*)	
	Read(*A*)	**Read**(*A*)	
	A := *A* − 100	*A* := *A* − 100	
	Write(*A*)	**Write**(*A*)	
	Lockx(*B*)	**Lockx**(*B*)	
	Unlock(*A*)	**Unlock**(*A*)	
	Read(*B*)	**Read**(*B*)	
	B := *B* + 100	*B* := *B* + 100	
	Write(*B*)	**Write**(*B*)	
↓	**Unlock**(*B*)	**Unlock**(*B*)	

some data-items locked even though the transactions no longer need these items. This extended locking forces a serialization of the two transactions and gives correct results.

12.4.1 Two-Phase Locking

The correctness of the schedules of Figures 12.14 and 12.15 and of the transactions in Figure 12.13 lead us to the observation that both these solutions involve transactions whose locking and unlocking operations are monotonic, in the sense that all locks are first acquired before any of the locks are released. Once a lock is released, no additional locks are requested. In other words, the release of the locks is delayed until all locks on all data-items required by the transaction have been acquired.

This method of locking is called **two-phase locking.** It has two phases, a **growing phase** wherein the number of locks increase from zero to the maximum for the transaction, and a **contracting phase** wherein the number of locks held decreases from the maximum to zero. Both of these phases are monotonic; the number of locks are only increasing in the first phase and decreasing in the second phase. Once a

Figure 12.15 Another solution to the inconsistent read problem.

	Schedule	Transaction T_{20}	Transaction T_{21}
	Lockx(*A*)	**Lockx**(*A*)	
	Read(*A*)	**Read**(*A*)	
	$A := A - 100$	$A := A - 100$	
	Write(*A*)	**Write**(*A*)	
T	**Lockx**(*B*)	**Lockx**(*B*)	
i	**Unlock**(*A*)	**Unlock**(*A*)	
m	**Read**(*B*)	**Read**(*B*)	
e	$B := B + 100$	$B := B + 100$	
	Write(*B*)	**Write**(*B*)	
	Unlock(*B*)	**Unlock**(*B*)	
	Lockx(*Sum*)		**Lockx***Sum*)
	$Sum := 0$		$Sum := 0$
	Locks(*A*)		**Locks**(*A*)
	Read(*A*)		**Read**(*A*)
	$Sum := Sum + A$		$Sum := Sum + A$
	Locks(*B*)		**Locks**(*B*)
	Read(*B*)		**Read**(*B*)
	$Sum := Sum + B$		$Sum := Sum + B$
	Write(*Sum*)		**Write**(*Sum*)
	Unlock(*B*)		**Unlock**(*B*)
	Unlock(*A*)		**Unlock**(*A*)
↓	**Unlock**(*Sum*)		**Unlock**(*Sum*)

transaction starts releasing locks, it is not allowed to request any further locks. In this way a transaction is obliged to request all locks it may need during its life before it releases any. This leads to a possible lower degree of concurrency.

The two-phase locking protocol ensures that the schedules involving transactions using this protocol will always be serializable. For instance, if S is a schedule containing the interleaved operations from a number of transactions, $T_1, T_2, \ldots, T_k$ and all the transactions are using the two-phase locking protocol, schedule S is serializable. This is because if the schedule is not serializable, the precedence graph for S will have a cycle made up of a subset of $\{T_1, T_2, \ldots, T_k\}$. Assume the cycle consists of $T_a \rightarrow T_b \rightarrow T_c \rightarrow \ldots T_x \rightarrow T_a$. This means that a lock operation by T_b is followed by an unlock operation by T_a; a lock operation by T_c is followed by an unlock operation by $T_b, \ldots$, and finally a lock operation by T_a is followed by an unlock operation by T_x. However, this is a contradiction of the assertion that T_a is using the two phase protocol. Thus the assumption that there was a cycle in the precedence graph is incorrect and hence S is serializable.

The transactions of Figure 12.13 use the two-phase locking protocol, and the schedules derived from the concurrent execution of these transactions given in Figures 12.14 and 12.15 are serializable. However, the transactions of Figure 12.11 do not follow the two-phase locking protocol and the schedule of Figure 12.12 is not serializable.

the **intention share mode** to indicate that the lower level is being locked in a share mode. Other transactions can access the node and all its lower levels, including the subtree being accessed by T_a; no transaction, however, can modify any portion of the database rooted at the node that was locked by T_a in the intention share mode. If transaction T_a intends to lock the lower level in the exclusive or share mode, then the ancestor is locked in the **intention exclusive mode** to indicate that the lower level is being locked in an exclusive or share mode. Another concurrent transaction, say T_b, needing to access any portion of this hierarchy in the exclusive or share mode can also lock this node in the intention exclusive mode. If T_b needs exclusive or share access to that portion of the subtree not being used by transaction T_a, it will place appropriate locks on it and can run concurrently with T_a. However, if T_b needs access to any portion of the subtree locked in the exclusive mode by T_a, then the explicit exclusive locks on these nodes will cause T_b to wait until T_a releases these explicit exclusive locks.

The intention lock locks a node to indicate that the lower level nodes are being locked either in the share or the exclusive mode, but it does no implicit locking of lower levels. Each lower level has to be locked explicitly in whichever mode required by the transaction. This adds a fairly large overhead if a transaction needs to access a subtree of the database and modify only a small portion of the subtree rooted at the intentionally locked node. The **share and intention exclusive mode** of locking is thus introduced. The share and intention exclusive mode differs from the other form of intention locking in so far as it implicitly locks all lower level nodes as well as the node in question. This mode allows access by other transactions to share that portion of the subtree not exclusively locked and gives higher concurrency than achievable with a simple exclusive lock. This avoids the overhead of locking the root node and all nodes in the path leading to the subtree to be modified in the intention exclusive mode, followed by locking the subtree to be modified in the exclusive mode. It is replaced by locking the root node in the share and intention exclusive mode (which will lock all descendants implicitly in the same mode), followed by locking the root node of the subtree to be modified in the exclusive mode.

We summarize below the possible modes in which a node of the database hierarchy could be locked and the effect of the locking on the descendants of the node. Figure 12.17 gives the relative privilege of these modes of locking. The exclusive mode has the highest privilege and the intention share mode has the lowest privilege.

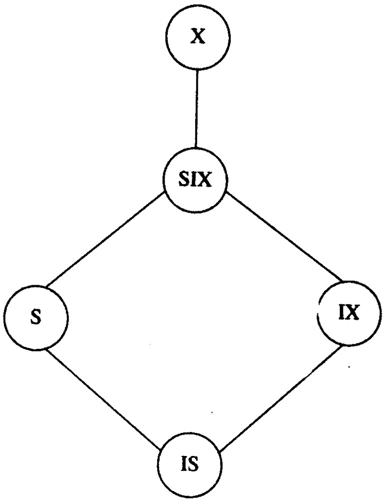
S or shared lock: The node in question *and implicitly all its descendants* are locked in the share mode; all these nodes, locked explicitly or implicitly, are accessible for read-only access. No transaction can update the node or any of its descendants when the node is locked in the shared mode.

X or exclusive lock: The node in question *and implicitly all its descendants* are exclusively locked by a single transaction. No other transaction can concurrently access these nodes.

IS or intention share: The node is locked in the intention share mode, which means that it or its descendants cannot be exclusively locked. The descendant of the node may be individually locked in a shared or intention shared mode. The descendants of the node that is locked in the IS mode are not locked implicitly.

IX or intention exclusive: The node is locked in an intention exclusive mode. This means that the node itself cannot be exclusively locked; however, any of the descendants, if not already locked, can be locked in any of the locking modes. The descendants of the node that is locked in the IX mode are not locked implicitly.

Figure 12.17 Relative privilege of the locking modes.



SIX or shared and intention exclusive: The node is locked in the shared and intention exclusive mode and all the descendants are implicitly locked in the shared mode. However, any of the descendants can be explicitly locked in the exclusive, intention exclusive, or shared and intention exclusive modes.

Relative Privilege of the Various Locking Modes

Figure 12.17 gives the relative privilege of the various modes of locking. The exclusive mode has the highest privilege: it locks out all other transactions from the portion of the database that is rooted at the node locked in the exclusive mode. All descendants of the node are implicitly locked in the exclusive mode. The intention share mode has the lowest privilege. The share mode is not comparable with the intention exclusive mode.

The advantage of the intention mode locking is that the lock manager knows that the lower level nodes of a node that is intentionally locked are or are being locked without having to examine all the lower level nodes. Furthermore, using the compatibility matrix shown in Figure 12.18 and discussed below, the lock manager can ascertain if a request for a lock can be granted.

Compatibility Matrix

Considering all the modes of locking described above, the compatibility between the current mode of locking for a node and the request of another transaction for locking the node in a given mode are given in Figure 12.18. The entry yes indicates that the request will be granted and the transaction can continue. The entry no indicates that the request cannot be granted and the requesting transaction will have to wait.

not be two phase and they are allowed to unlock an item before locking another item. The only requirement is that the transaction must have a lock on the parent of the node being locked and that the item was not previously locked by the transaction.

Consider the database of Figure 12.19. A transaction, for instance T_a, can start off by locking the entire database. Then it proceeds to lock Portion_1, Record Type_1 and Record Type_2. At this point it unlocks the database and then locks record occurrences R_{11} and R_{21}, followed by unlocking Portion_1, and Record Type_2. Another transaction, T_b, can then proceed by first locking Record Type_2 followed by locking record occurrences R_{22}. The first transaction can now lock record occurrence R_{12}.

The advantage of the tree-locking protocol over the two-phase locking protocol is that a data-item can be released earlier by a transaction if the data-item (and of course, any of its yet unlocked descendants in the subtree rooted at the data-item) is not required by the transaction. In this way a greater amount of concurrency is feasible. However, since a descendant is not locked by the lock on a parent, the number of locks and associated locking overhead, including the resulting waits, is increased.

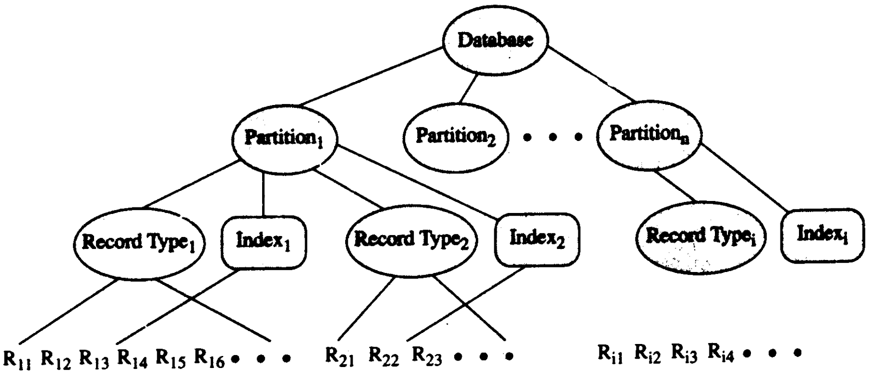
12.4.5 DAG Database Storage Structure

The use of indexes to obtain direct access to the records of the database causes the hierarchical storage structure to be converted into a **directed acyclic graph (DAG)** as shown in Figure 12.20. The locking protocol can be extended to a DAG structure; the only additional rule is that to lock a node in the IX, SIX, or X modes, all the parents of the node have to be locked in a compatible mode that is at least an IX mode. Thus, no other transaction can get a lock to any of the parents in the S, SIX, or X modes. This is illustrated in Example 12.5.

Example 12.5

To add a record occurrence to the Record Type_1, which uses an index_1 for direct access to the records, the sequence of locking is as follows: (1) lock the database in the IX mode, (2) lock Portion_1 in the IX mode, (3) lock Record Type_1 and index_1 in the X mode. With this method of locking, the phantom phenomenon is avoided at the expense of lower concurrency. ■

Figure 12.20 Sample DAG database storage structure.



As in the case of two-phase locking, deadlock is possible in the locking scheme using hierarchical granularity of locking. Additional details regarding references to techniques to reduce and eliminate such deadlock are cited in the bibliographic notes.

12.5 Timestamp-Based Order

In the timestamp-based method, a serial order is created among the concurrent transaction by assigning to each transaction a unique nondecreasing number. The usual value assigned to each transaction is the system clock value at the start of the transaction, hence the name **timestamp ordering.** A variation of this scheme that is used in a distributed environment includes the site of a transaction appended to the system-wide clock value. This value can then be used in deciding the order in which the conflict between two transactions is resolved. A transaction with a smaller timestamp value is considered to be an "older" transaction than another transaction with a larger timestamp value.

The serializability that the system enforces is the chronological order of the timestamps of the concurrent transactions. If two transaction T_i and T_j with the time stamp values t_i and t_j respectively, such that $t_i < t_j$, are to run concurrently, then the schedule produced by the system is equivalent to running the older transaction T_i first, followed by the younger one, T_j.

The contention problem between two transactions in the timestamp ordering system is resolved by rolling back one of the conflicting transactions. A conflict is said to occur when an older transaction tries to read a value that is written by a younger transaction or when an older transaction tries to modify a value already read or written by a younger transaction. Both of these attempts signify that the older transaction was "too late" in performing the required read/write operations and it could be using values from different "generations" for different data-items.

In order for the system to determine if an older transaction is processing a value already read by or written by a younger transaction, each data-item has, in addition to the value of the item, two timestamps: a **write timestamp** and a **read timestamp.** Data-item X is thus represented by a triple X: $\{x, W_x, R_x\}$ where each component of the triple is interpreted as given below:

x, the value of the data-item X

W_x, the write timestamp value, the largest timestamp value of any transaction that was allowed to write a value of X.

R_x, the read timestamp value, the largest timestamp value of any transaction that was allowed to read the current value X.

Now let us see how these timestamp values find their way into the data structure of a data-item and how all these values are modified. A transaction T_a with the timesteamp value of t_a issues a read operation for the data-item X with the values $\{x, W_x, R_x\}$.

- This request will succeed if $t_a \geq W_x$, since transaction T_a is younger than the transaction that last wrote (or modified) the value of X. Transaction T_a is

After step 5	*A*: 400, W_a, t_{23}	*B*: 500, W_b, R_b
After step 7	*A*: 300, t_{23}, t_{23}	*B*: 500, W_b, R_b
After siep 8	*A*: 300, t_{23}, t_{23}	*B*: 500, W_b, t_{22}
After step 1C	the value displayed will be 900	
After step 12	*A*: 300, t_{23}, t_{23}	*B*: 500, W_b, t_{23}
After step 14	*A*: 300, t_{23}, t_{23}	*B*: 600, t_{23}, t_{23}

After step 14 the value displayed will be 900 ■

In the following example we illustrate a scnedule where the older transaction is rolled back.

Example 12.7

Figure F Senalizable schedule produced after a rollback.

Step	Schedule	Transaction T_{22}	Transaction T_{23}
1	*Sum* := 0	*Sum* := 0	
2	*Sum* := 0		*Sum* := 0
3	**Read**(*A*)		**Read**(*A*)
4	*A* := *A* − 100		*A* := *A* − 100
5	**Write**(*A*)		**Write**(*A*)
6	**Read**(*A*)	**Read**(*A*)* causes a rollback of T_{22}	
7	*Sum* := *Sum* + *A*		*Sum* := *Sum* + *A*
8	**Read**(*B*)		**Read**(*B*)
9	*B* := *B* + 100		*B* := *B* + 100
10	**Write**(*B*)		**Write**(*B*)
11	*Sum* := *Sum* + *B*		*Sum* := *Sum* + *B*
12	**Show**(*Sum*)		**Show**(*Sum*)
13	*Sum* := 0	*Sum* := 0 with a timestamp $t_{22}'(> t_{23})$	
14	**Read**(*A*)	**Read**(*A*)	
15	*Sum* := *Sum* + *A*	*Sum* := *Sum* + *A*	
16	**Read**(*B*)	**Read**(*B*)	
17	*Sum* := *Sum* + *B*	*Sum* := *Sum* + *B*	
18	**Show**(*Sum*)	**Show**(*Sum*)	

Consider the schedule shown in Figure F. Transaction T_{22} is rolled back and rerun after step 6. When it is rolled back, a new timestamp value t_{22}' which would be greater than t_{23}, is assigned to it. The sequence of changes is given below:

Initially	*A*: 400, W_a, R_a	*B*: 500, W_b, R_b
After step 3	*A*: 400, W_a, t_{23}	*B*: 500, W_b, R_b
After step 5	*A*: 300, t_{23}, t_{23}	*B*: 500, W_b, R_b
After step 6	*A*: 300, t_{23}, t_{23}	*B*: 500, W_b, R_b*

(*causes a rollback of T_{22} which would be reassigned a new timestamp (t_{22}^{1}, > t_{23}) and would be reexecuted)

After step 8 A: 300, t_{23}, t_{23} B: 500, W_b, t_{23}
After step 10 A: 300, t_{23}, t_{23} B: 600, t_{23}, t_{23}
After step 12 the value displayed will be 900
After step 14 A: 300, t_{23}, t_{22}' B: 600, t_{23}, t_{23}
After step 16 A: 300, t_{23}, t_{22}' B: 600, t_{23}, t_{22}'
After step 18 the value displayed will be 900 ■

Example 12.8 below illustrates a case where the write operation of a transaction could be ignored.

Example 12.8

In the example illustrated in Figure G, we have three transactions, T_{24}, T_{25}, and T_{26} with timestamp values of t_{24}, t_{25}, and t_{26} respectively ($t_{24} < t_{25} < t_{26}$). Note that transactions T_{24} and T_{26} are write-only with respect to data-item B.

Figure G Another serializable schedule.

Step	Schedule	Transaction T_{24}	Transaction T_{25}	Transaction T_{26}
1	**Read**(A)	**Read**(A)		
2	$A := A + 1$	$A := A + 1$		
3	**Write**(A)	**Write**(A)		
4	**Read**(C)		**Read**(C)	
5	$C := C * 3$		$C := C * 3$	
6	**Read**(C)			**Read**(C)
7	**Write**(C		**Write**(C)* causes a rollback of transaction T_{25}	
8	$C := C * 2$			$C := C * 2$
9	**Write**(C)			**Write**(C)
10	$B := 100$			$B := 100$
11	**Write**(B)			**Write**(B)
12	$B := 150$	$B := 150$		
13	**Write**(B)	**Write**(B)** causes the write operation to be ignored		
14	**Read**(C)		**Read**(C)	
15	$C := C * 3$		$C := C * 3$	
16	**Write**(C)		**Write**(C)	

Initially	A: 10, W_a, R_a	B: 50, W_b, R_b	C: 5, W_c, R_c
After step 1	A: 10, W_a, t_{24}	B: 50, W_b, R_b	C: 5, W_c, R_c
After step 3	A: 11, t_{24}, t_{24}	B: 50, W_b, R_t	C: 5, W_c, R_c
After step 4	A: 11, t_{24}, t_{24}	B: 50, W_b, R_b	C: 5, W_c, t_{25}
After step 5	A: 11, t_{24}, t_{24}	B: 50, W_b, R_t	C: 5, W_c, t_{25}

The write timestamp of a version of a data-item is the timestamp value of the transaction that wrote the version of the data-item. In other words, a value of the data-item X with the write timestamp value W_x was written by a transaction with a timestamp value of W_x. Note, that here we are ignoring the time lapse from the start of the transaction to the generation of the new version. The timestamps are in reality pseudotimes and a nondecreasing counter can be used instead of a timestamp with similar results.

The read timestamp of a version of a data-item is the timestamp value of the most recent transaction that successfully read the version of the data-item. A version of the data-item with the read timestamp of R_x was read by a transaction with a timestamp value of R_x. The read timestamp value is the same as the time of modification of the value of the data-item, if another version of the data-item exists; otherwise it remains the most recent version of the data-item. This is because a new' version usually will not be created without first reading the current most recent version.

If a transaction T_i with a time stamp value of t_i writes a value x_i for the kth version of a data-item X, then the kth version of X will have the value x_i. W_{xk}, the write timestamp value, and R_{xk}, the read timestamp value of X_k will both be initialized to t_i.

A transaction needing to read the value of data-item X is directed to read that version of X that was the most recent version, with respect to the timestamp ordering of the transaction. We call this version the **relative-most-recent version.** Thus, if a transaction T_a with the timestamp value of t_a needs to read the value of data-item X, it will read the version X_j such that W_{xj} is the largest write timestamp value of all versions of X that is less than or equal to t_a. The read timestamp value of version X_j of X, read by transaction T_a, is updated to t_a if $t_a > R_{xj}$.

A transaction T_a, wanting to modify a data-item value will first read the relative-most-recent version X_j of data-item X. When it tries to write a new value of X, one of the following actions will be performed:

- A new version of X, e.g., version X_j', is created and stored with the value x_j and with the timestamp values of $W_{xj}' = R_{xj}' = t_a$, if the current value of $R_{xj} \leq t_a$. This ensures that transaction T_a was the most recent transaction to read the value of version X_j, and no other transaction has read the value that was the basis of updating by T_a.
- Transaction T_a is aborted and rolled back if the current value of $R_{xj} > t_a$. The reason is that another younger transaction has read the value of version X_j and may have used it and/or modified it. Transaction T_a was too late and it should try to rerun to obtain the current most recent version of the value of X.

It is easy to see that the value of the write timestamp is the same as the time of generation of a new version of the value of a data-item, and the read timestamp value is the same as the time of modification of the value of the data-item.

A transaction T_a with a timestamp value of t_a, writing a new version of a data-item X without first reading, creates a new version of X with the write timestamp and read timestamp values of t_a.

It can be shown that any schedule generated according to the above requirements is serializable, and the result obtained by a set of concurrent transactions is the same as that obtained by some serial execution of the set with a single version of the data-items.

Example 12.10

Consider the schedule given in Figure I for two concurrent transactions T_{22} and T_{23} of Figure 12.21. Suppose the multiversion technique is used for concurrency control. Assume initially that a single version exists for data-items *A* and *B* with their initial values being:

A: {{ 400, W_a, R_a}} and *B:* {{ 500, W_b, R_b}}

Transaction T_{22} has a timestamp value of t_{22}; transaction T_{23} has a timestamp value of t_{23}.

$t_{22} < t_{23}$, $W_a < t_{22}$, $R_a < t_{22}$, $W_b < t_{22}$, $R_b < t_{22}$

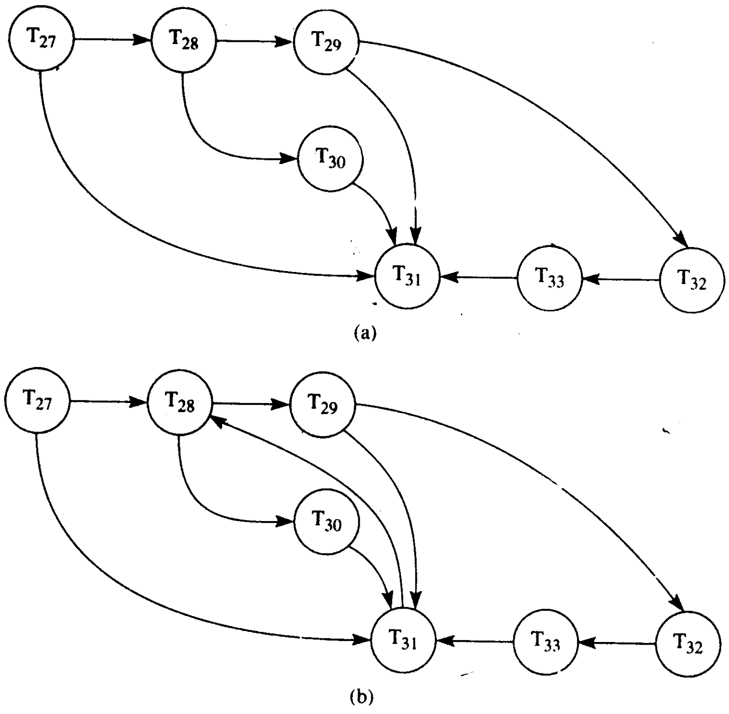
The modifications after the following steps are:

After step 3	*A:* {{400, W_a, t_{23}}}
	B: {{500, W_b, R_b}}
After step 5	*A:* {{400, W_a, t_{23}}, {300, t_{23}, t_{23}}}
	B: {{500, W_b, R_b}}
After step 6	*A:* {{400, W_a, t_{23}}, {300, t_{23}, t_{23}}}
	B: {{ 500, W_b, R_b}}
After step 8	*A:* {{400, W_a, t_{23}}, {300, t_{23}, t_{23}}}
	B: {{500, W_b, t_{22}}}
After step 10	the value shown by T_{22} is 900
After step 12	*A:* {{400, W_a, t_{23}}, {300, t_{23}, t_{23}}}
	B: {{500, W_b, t_{23}}}
After step 14	*A:* {{400, W_a, t_{23}}, {300, t_{23}, t_{23}}}
	B: {{ 500, W_b, t_{23}}, {600, t_{23}, t_{23}}}
After step 16	the value shown by T_{23} is 900

Figure I Schedule for the multiversion technique.

Step	Schedule	Transaction T_{22}	Transaction T_{23}
1	*Sum* := 0	*Sum* := 0	
2	*Sum* := 0		*Sum* := 0
3	**Read**(*A*)		**Read**(*A*)
4	*A* := *A* − 100		*A* := *A* − 100
5	**Write**(*A*)		**Write**(*A*)
6	**Read**(*A*)	**Read**(*A*)	
7	*Sum* := *Sum* + *A*	*Sum* := *Sum* + *A*	
8	**Read**(*B*)	**Read**(*B*)	
9	*Sum* := *Sum* + *B*	*Sum* := Sum + B	
10	**Show**(*Sum*)	**Show**(*Sum*)	
11	*Sum* := *Sum* + *A*		*Sum* := *Sum* + *A*
12	**Read**(*B*)		**Read**(*B*)
13	*B* := *B* + 100		*B* := *B* + 100
14	**Write**(*B*)		**Write**(*B*)
15	*Sum* := *Sum* + *B*		*Sum* := *Sum* + *B*
16	**Show**(*Sum*)		**Show**(*Sum*) ■

Figure 12.22 Wait-for graph showing (a) no cycle and hence no deadlock; (b) a cycle and hence a deadlock.



been satisfied, adds the arc from the node for transaction T_{31} to the node for transaction T_{28}. The addition of this arc causes the wait-for graph to have a number of cycles. One of these cycles is indicated by the arc from transaction T_{28} to transaction T_{30}, then, from transaction T_{30} to T_{31}, and finally from T_{31} back to T_{28}. Consequently part b represents a situation where a number of sets of transactions are deadlocked.

Since a cycle in the wait-for graph is a necessary and sufficient condition for deadlock to exist, the deadlock detection algorithm generates the wait-for graph at regular intervals and examines it for a chain. If the interval chosen is very small, deadlock detection will add considerable overhead; if the interval chosen is very large, there is a possibility that a deadlock will not be detected for a long period. The choice of interval depends on the frequency of deadlocks and the cost of not detecting the deadlocks for the chosen interval. The overhead of keeping the wait-for graph continuously, adding arcs as requests are blocked and removing them as locks are given up, would be very high.

The deadlock detection algorithm is given on page 598. In this algorithm we use a table called Wait_for table. It contains columns for each of the following: transaction IDs; the data-items for which they have acquired a lock; and the data-items they are waiting for (these wait-for items are currently locked in an incompatible mode by other transactions). The algorithm starts with the assumption that there is no deadlock. It locates a transaction, T_s, which is waiting for a data-item. If the data-item is currently locked by transaction T_r, the latter is in the wait-for graph. If

T_r in turn is waiting for a data item currently locked by transaction T_p, this transaction is also in the wait-for graph. In this way the algorithm finds all other transactions involved in a wait-for graph starting with transaction T_s. If the algorithm finally finds that there is a transaction T_q waiting for a data-item currently locked by T_s, the wait-for graph leads back to the starting transaction. Consequently the algorithm concludes that a cycle exists in the wait-for graph and there is a potential deadlock situation.

Example 12.11 Consider the wait-for table of Figure J. The wait-for graph for the transactions in this chain is given by Figure 12.22a. It has no cycles and hence there are no deadlocks. However, if transaction T_{31} makes a request for data-item C, the wait-for graph is converted into the one given in Figure 12.22b. This graph has a cycle that starts at transaction T_{28}, goes through transactions T_{30}, T_{31}; and back to T_{28}, and Algorithm 12.2 detects it. There are other cycles as well.

Figure J Wait-for table for Example 12.11.

Transaction_Id	Data_items_locked	Data_items_waiting_for
T_{27}	*B*	*C, A*
T_{28}	*C, M*	*H, G*
T_{29}	*H*	*D, E*
T_{30}	*G*	*A*
T_{31}	*A, E*	*(C)*
T_{32}	*D, I*	*F*
T_{33}	*F*	*E*

■

An adaptive system may initially choose a fairly infrequent interval to run the deadlock detection algorithm. Every time a deadlock is detected, the deadlock detection frequency could be increased, for example, to twice the previous frequency and every time no deadlock is detected, the frequency could be reduced, for example, to half the previous frequency. Of course an upper and lower limit to the frequency would have to be established.

Recovery from Deadlock

To recover from deadlock, the cycles in the wait-for graph must be broken. The common method of doing this is to roll back one or more transactions in the cycles

Algorithm 12.2 Deadlock Detection

Input and Data Structure Used: A table called Wait_for_Table that contains: transaction IDs, the data-items they have acquired a lock on, and the data-items they are waiting for (these wait-for items are currently locked in an incompatible mode by other transactions). A Boolean variable Deadlock_situation. A first-in, first-out stack, Transaction_stack, to hold transaction IDs: this stack will contain the transactions in a deadlocked chain if a deadlock is detected.

Output Whether the system is deadlocked and if so, the transactions in the cycle.

```
Initialize Deadlock_Situation to false;
Initialize Transaction_stack to empty;
for next transaction in table while not Deadlock_Situation
  begin
   Push next Transaction ID into Transaction_stack ;
    for next Data_item_waiting_for of
        transaction on top of Transaction_stack and
                        while not Deadlock_Situation
                              and not Transaction_stack empty
     begin
        D_next := next Data_item_waiting_for
        find Tran_ID of transaction which has locked D_next
        if Tran_ID is in stack
         then Deadlock_Situation := true
         else Push Tran_ID to Transaction_stack
        end
   Pop Transaction_stack
   end
```

until the system exhibits no further deadlock situation. The selection of the transactions to be rolled back is based on the following considerations:

- The progress of the transaction and the number of data-items it has used and modified. It is preferable to roll back a transaction that has just started or has not modified any data-item, rather than one that has run for a considerable time and/or has modified many data-items.
- The amount of computing remaining for the transaction and the number of data-items that have yet to be accessed by the transaction. It is preferable not to roll back a transaction if it has almost run to completion and/or it needs very few additional data-items before its termination.
- The relative cost of rolling back a transaction. Notwithstanding the above considerations, it is preferable to roll back a less important or noncritical transaction.

Once the selection of the transaction to be rolled back is made, the simplest scenario consists of rolling back the transaction to the start of the transaction, i.e., abort the transaction and restart it, *de nouveau*. If, however, additional logging is done by the system to maintain the state of all active transactions, the rollback need not be total, merely far enough to break the cycle indicating the deadlock situation. Nonetheless, this overhead may be excessive for many applications.

The process of deadlock recovery must also ensure that a given transaction is not continuously the one selected for rollback. If this is not avoided, the transaction will never (or at least for a period that looks like never) complete. This is starving a transaction!

12.8.2 Deadlock Avoidance

In the deadlock avoidance scheme, care is taken to ensure that a circular chain of processes holding some resources and waiting for additional ones held by other transactions in the chain never occurs. The two-phase locking protocol ensures serializability, but does not ensure a deadlock-free situation. This is illustrated in Example 12.12.

Example 12.12

Consider transactions T_{34} and T_{35} given in Figure K and the schedule of Figure L. These are two-phase transactions; however, a deadlock situation exists in Figure L, as transaction T_{34} waits for a data-item held by transaction T_{35}; later on, transaction T_{35} itself waits for a data-item held by T_{34}, which is already blocked from further progress.

Figure K Two-phase transactions.

Transaction T_{34}	Transaction T_{35}
Sum := 0	*Sum* := 0
Locks(*A*)	**Lockx**(*B*)
Read(*A*)	**Read**(*B*)
Sum := *Sum* + *A*	*B* := *B* + 100
Locks(*B*)	**Write**(*B*)
Read(*B*)	*Sum* := *Sum* + *B*
Sum := *Sum* + *B*	**Lockx**(*A*)
Show(*Sum*)	**Unlock**(*B*)
Unlock(*A*)	**Read**(*A*)
Unlock(*B*)	*A* := A − 100
	Write(*A*)
	Unlock(*A*)
	Sum := *Sum* + *A*
	Show(*Sum*)

Figure L Schedule leading to deadlock with two-phase transactions.

	Schedule	Transaction T_{34}	Transaction T_{35}
	Sum := 0	*Sum* := 0	
	Locks(*A*)	**Locks**(*A*)	
	Read(*A*;	**Read**(*A*)	
	Sum := *Sum* + *A*	*Sum* := *Sum* + *A*	
T	*Sum* := 0		*Sum* := 0
i	**Lockx**(*B*)		**Lockx**(*B*)
m	**Read**(*B*)		**Read**(*B*)
e	*B* := *B* + 100		*B* := *B* + 100
	Write(*B*)		**Write**(*B*)
	Sum := *Sum* + *B*		*Sum* := *Sum* + *B*
	Locks(*B*)	**Locks**(*B*)* transaction T_{34} will wait	
	Lockx(*A*)		**Lockx**(*A*)* T_{35} will wait

■

One of the simplest methods of avoiding a deadlock situation is to lock all data-items at the beginning of a transaction. This has to be done in an atomic manner, otherwise there could be a deadlock situation again. The main disadvantage of this scheme is that the degree of concurrency is lowered considerably. A transaction typically needs a given data-item for a very short interval. Locking all data-items for the entire duration of a transaction makes these data-items inaccessible to other concurrent transactions. This could be the case even though the transaction holding a lock on these data-items may not need them for a long time after it acquires a lock on them.

Another approach used in avoiding deadlock is assigning an order to the data-items and requiring the transactions to request locks in a given order, such as only ascending order. Thus, data-items may be ordered as having rank 1, 2, 3, and so on. A transaction T requiring data-items *A* (with a rank of i) and *B* (with a rank of j with j > i) must first request a lock for the data-item with the lowest rank, namely *A*. When it succeeds in getting the lock for *A*, only then can it request a lock for data-item *B*. All transactions follow such a protocol, even though within the body of the transaction the data-items are not required in the same order as the ranking of the data-items for lock requests. This scheme reduces the concurrency, but not to the same extent as the first scheme.

Another set of approaches to deadlock avoidance is to decide whether to wait or abort and roll back a transaction, if a transaction finds that the data-item it requests is locked in an incompatible mode by another transaction. The decision is controlled by timestamp values. Aborted and rolled back transactions retain their timestamp values and hence their "seniority." So, in subsequent situations, they would eventually get a "higher priority." We examine below two such approaches called wait-die and wound-wait.