UNIT-5

Concurrency Control

Concurrency control is the procedure in DBMS for managing simultaneous operations without conflicting with each another. Concurrent access is quite easy if all users are just reading data. There is no way they can interfere with one another. Though for any practical database, would have a mix of reading and WRITE operations and hence the concurrency is a challenge.

Concurrency control is used to address such conflicts which mostly occur with a multi-user system. It helps you to make sure that database transactions are performed concurrently without violating the data integrity of respective databases.

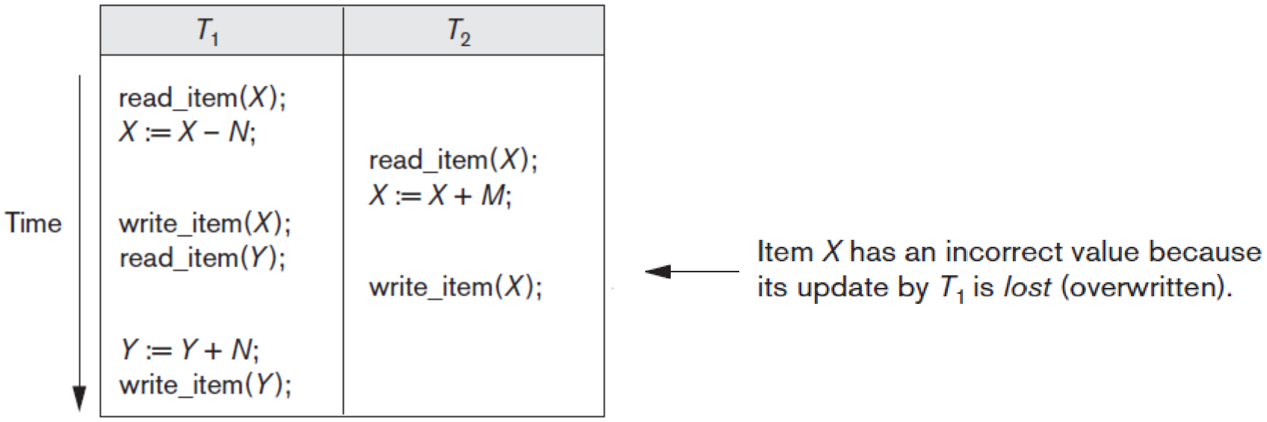
Therefore, concurrency control is a most important element for the proper functioning of a system where two or multiple database transactions that require access to the same data, are executed simultaneously.

Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncontrolled manner.

1. The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

Suppose that transactions *T*1 and *T*2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure; then the final value of item *X* is incorrect because *T*2 reads the value of *X before T*1 changes it in the database, and hence the updated value resulting from *T*1 is lost. For example, if *X* = 80 at the start (originally there were 80 reservations on the flight), *N* = 5 (*T*1 transfers 5 seat reservations from the flight corresponding to *X* to the flight corresponding to *Y*), and *M* = 4 (*T*2 reserves 4 seats on *X*), the final result should be *X* = 79.However, in the interleaving of operations shown in Figure, it is *X* = 84 because the update in *T*1 that removed the five seats from *X* was *lost*.



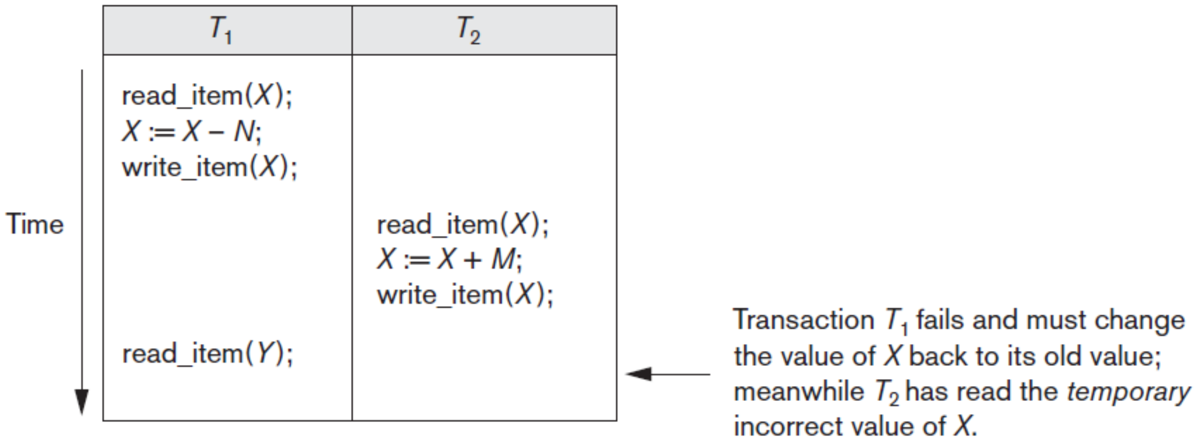
T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

2. The Temporary Update (or Dirty Read) Problem. This problem occurs when one transaction updates a database item and then the transaction fails for some reason. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure shows an example where *T*1 updates item *X* and then fails before completion, so the system must change *X* back to its original value. Before it can do so, however, transaction *T*2 reads the *temporary* value of *X*, which will not be recorded permanently in the database because of the failure of *T*1. The value of item *X* that is read by *T*2 is called *dirty data* because it has been created by a transaction that has not completed and committed yet; hence, this problem is also known as the *dirty read problem*.



T_1	T_2

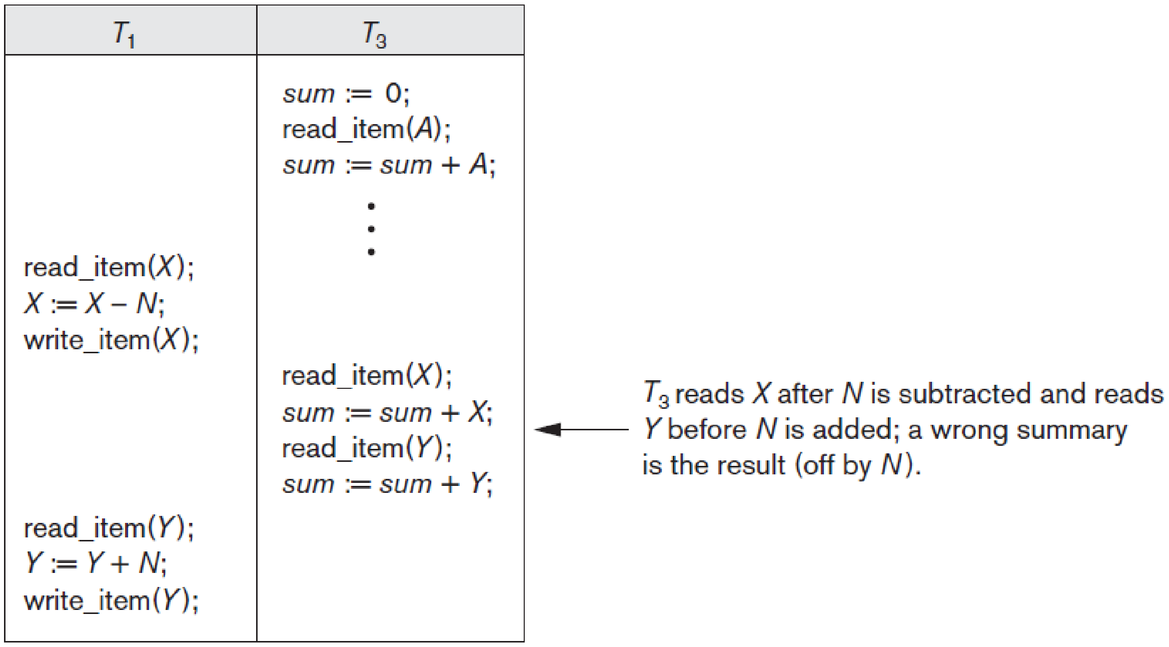
read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

3. The Incorrect Summary Problem. If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction *T*3 is calculating the total number of reservations on all the flights; meanwhile, transaction *T*1 is executing. If the interleaving of operations shown in Figure occurs, the result of *T*3 will be off by an amount *N* because *T*3 reads the value of *X after N* seats have been subtracted from it but reads the value of *Y before* those *N* seats have been added to it.



T_1	T_3

	sum := 0;
	read_item(A);
	sum := *sum* + A;
	⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	sum := *sum* + X;
	read_item(Y);
	sum := *sum* + Y;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

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

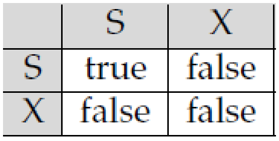
There are various modes in which a data item may be locked. We have two modes of locks:

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.

Given a set of lock modes, we can define a **compatibility function** on them as follows. Let *A* and *B* represent arbitrary lock modes. Suppose that a transaction *Ti* requests a lock of mode *A* on item *Q* on which transaction *Tj* (*Ti != Tj*) currently holds a lock of mode *B*. If transaction *Ti* can be granted a lock on *Q* immediately, in spite of the presence of the mode *B* lock, then we say mode *A* is **compatible** with mode *B*. Such a function can be represented conveniently by a matrix. The compatibility relation between the two modes of locking appears in the matrix. An element comp(*A*, *B*) of the matrix has the value *true* if and only if mode *A* is compatible with mode *B*.

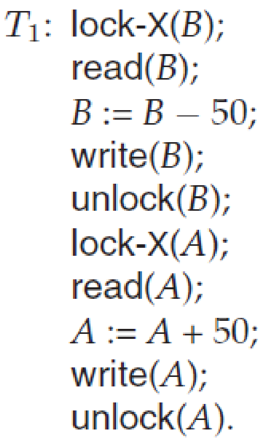


	S	X
S	true	false
X	false	false

Lock-compatibility matrix comp.

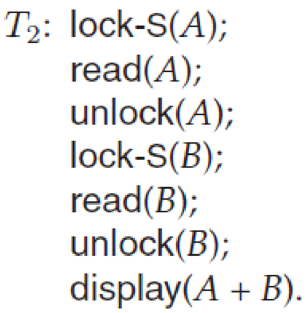
A transaction requests a shared lock on data item *Q* by executing the lock-S(*Q*) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction. A transaction can unlock a data item *Q* by the unlock(*Q*) instruction.

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.



```
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).
```

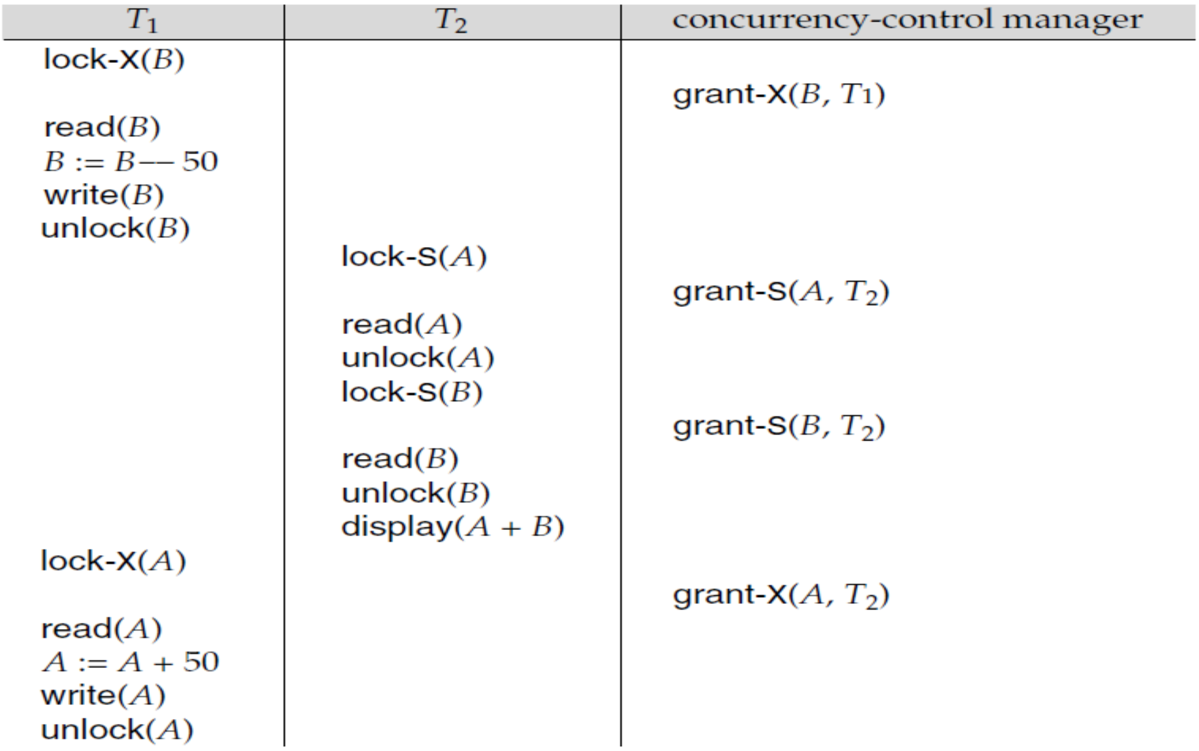
Transaction T_1



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

Transaction T_2

Let *A* and *B* be two accounts that are accessed by transactions *T*1 and *T*2. Transaction *T*1 transfers $50 from account *B* to account *A*. Transaction *T*2 displays the total amount of money in accounts *A* and *B*—that is, the sum *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*)	

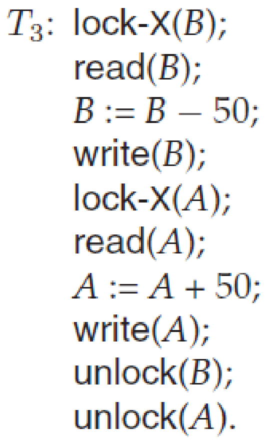
lock-X(*A*)		
		grant-X(*A*, T_2)

read(*A*)
A := *A* + 50
write(*A*)
unlock(*A*)		

Schedule-1

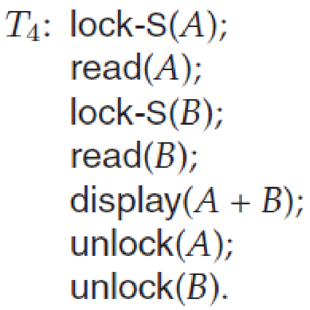
Suppose that the values of accounts *A* and *B* are $100 and $200, respectively. If these two transactions are executed serially, either in the order *T*1, *T*2 or the order *T*2, *T*1, then transaction *T*2 will display the value $300. If, however, these transactions are executed concurrently, then schedule 1 is possible. In this case, transaction *T*2 displays $250, which is incorrect. The reason for this mistake is that the transaction *T*1 unlocked data item *B* too early, as a result of which *T*2 saw an inconsistent state.

Suppose now that unlocking is delayed to the end of the transaction. Transaction T3 corresponds to T1 with unlocking delayed Transaction and T4 corresponds to T2 with unlocking delayed.



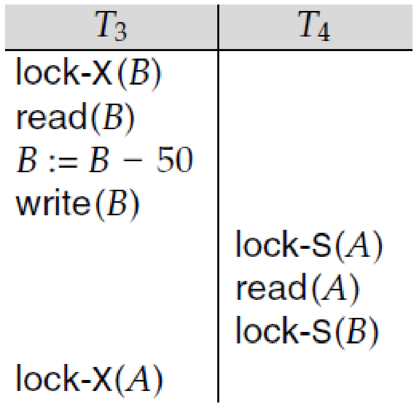
```
T3: lock-X(B);
    read(B);
    B := B − 50;
    write(B);
    lock-X(A);
    read(A);
    A := A + 50;
    write(A);
    unlock(B);
    unlock(A).
```

Transaction T_3



```
T4: lock-S(A);
    read(A);
    lock-S(B);
    read(B);
    display(A + B);
    unlock(A);
    unlock(B).
```

Transaction T_4



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)	

Schedule-2

In schedule 2, transactions T_3 and T_4 will not show incorrect result. T_4 will not print an inconsistent result in any of the possible schedules of T_3 and T_4.

Unfortunately, locking can lead to an undesirable situation. Consider the partial schedule for T3 and T4. Since T3 is holding an exclusive-mode lock on B and T4 is requesting a shared-mode lock on B, T4 is waiting for T3 to unlock B. Similarly, since T4 is holding a shared-mode lock on A and T3 is requesting an exclusive-mode lock on A, T3 is waiting for T4 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.

Deadlocks are a necessary evil associated with locking, if we want to avoid inconsistent states. Deadlocks are definitely preferable to inconsistent states, since they can be handled by rolling back of transactions.

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.

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 starvation.

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

1. There is no other transaction holding a lock on Q in a mode that conflicts with M.

2. 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.

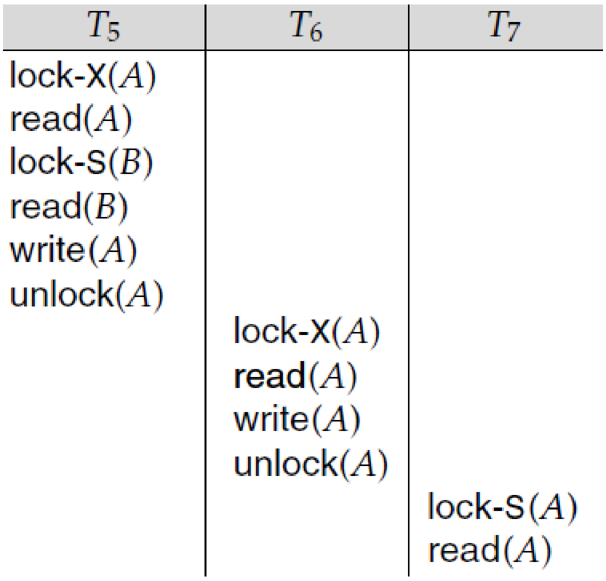
Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.

For example, transactions T3 and T4 are two phase. On the other hand, transactions T1 and T2 are not two phase.

The two-phase locking protocol ensures conflict serializability. Transaction can be ordered according to their lock points. 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.

Two-phase locking does not ensure freedom from deadlock. Observe that transactions T3 and T4 are two phase, but, in schedule 2, they are deadlocked.

Cascading rollback may occur under two-phase locking.



T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Schedule-3

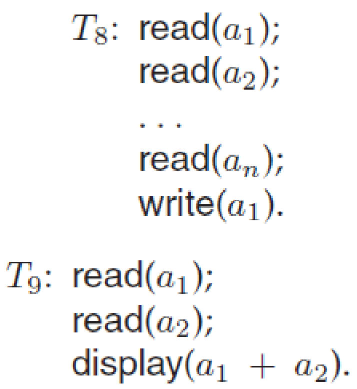
Each transaction observes the two-phase locking protocol, but the failure of T5 after the read(A) step of T7 leads to cascading rollback of T6 and T7.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking

There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol.

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. With rigorous two-phase locking, transactions can be serialized in the order in which they commit.

Consider the following two transactions, for which we have shown only some of the significant read and write operations:

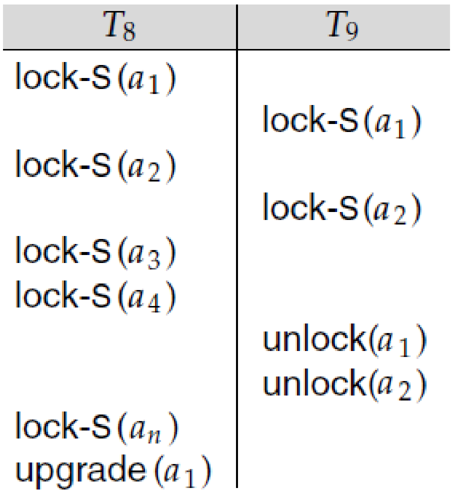


```
$T_8$: read($a_1$);
       read($a_2$);
       . . .
       read($a_n$);
       write($a_1$).

$T_9$: 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.



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)	

Schedule-4 with lock conversion

Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

Timestamp-Based Protocols

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:

- **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

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 Rtimestamp(*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.

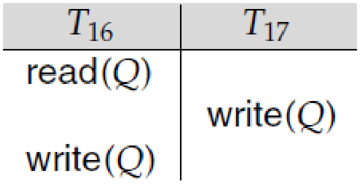
b. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, the system rejects this write operation and rolls *Ti* back.

c. Otherwise, the system executes the write operation and sets W-timestamp(*Q*) to TS(*Ti*).

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.

Thomas' Write Rule



T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

Schedule-5

Let us consider schedule-5 and apply the timestamp-ordering protocol. Since T16 starts before T17, we shall assume that TS(T16) < TS(T17). The read(Q) operation of T16 succeeds, as does the write(Q) operation of T17.When T16 attempts its write(Q) operation, we find that TS(T16) < W-timestamp(Q), since W-timestamp(Q) = TS(T17). Thus, the write(Q) by T16 is rejected and transaction T16 must be rolled back.

Although the rollback of T16 is required by the timestamp-ordering protocol, it is unnecessary. Since T17 has already written Q, the value that T16 is attempting to write is one that will never need to be read.

This observation leads to a modified version of the timestamp-ordering protocol in which obsolete write operations can be ignored under certain circumstances.

The modification to the timestamp-ordering protocol, called **Thomas' write rule**, is this:

Suppose that transaction *Ti* issues write(*Q*).
1. If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was previously needed, and it had been assumed that the value would never be produced. Hence, the system rejects the write operation and rolls *Ti* back.
2. If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*. Hence, this write operation can be ignored.
3. Otherwise, the system executes the write operation and sets W-timestamp(*Q*) to TS(*Ti*).

Validation-Based Protocols

The validation based protocols require 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*.

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*:

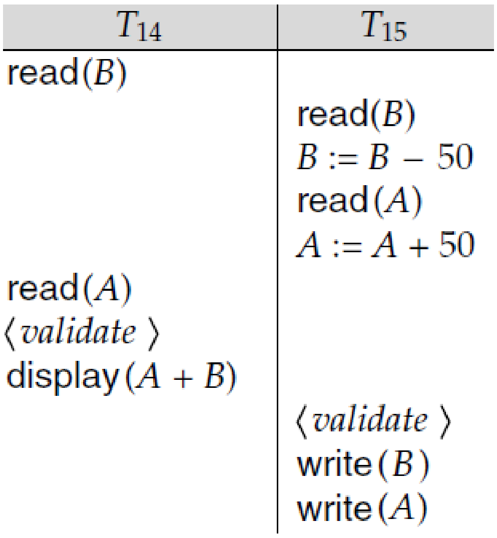
1. Start(*Ti*), the time when *Ti* started its execution.
2. Validation(*Ti*), the time when *Ti* finished its read phase and started its validation phase.
3. Finish(*Ti*), the time when *Ti* finished its write phase.

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:

1. Finish(*Ti*) < Start(*Tj*). Since *Ti* completes its execution before *Tj* started, the serializability order is indeed maintained.

2. 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)

Schedule-6, a schedule produced by validation

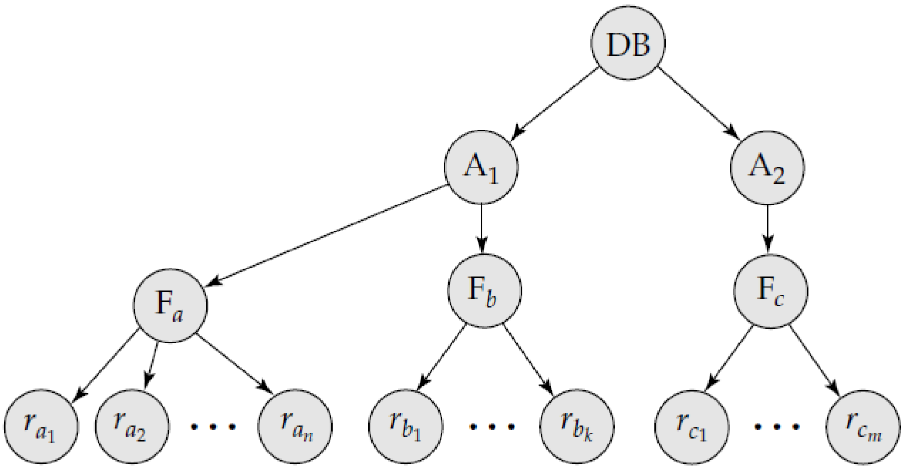
Suppose that TS(T_{14}) < TS(T_{15}). Then, the validation phase succeeds in the schedule 6. Note that the writes to the actual variables are performed only after the validation phase of T_{15}. Thus, T_{14} reads the old values of *B* and *A*, and this schedule is serializable.

The validation scheme automatically guards against cascading rollbacks. However, there is a possibility of starvation of long transactions.
This validation scheme is called the **optimistic concurrency control** scheme since transactions execute optimistically.

Multiple Granularity

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed. There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit.

For example, if a transaction *Ti* needs to access the entire database, and a locking protocol is used, then *Ti* must lock each item in the database. Clearly, executing these locks is time consuming. It would be better if *Ti* could issue a *single* lock request to lock the entire database. On the other hand, if transaction *Tj* needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost.



Granularity Hierarchy

We needed a mechanism to allow the system to define multiple levels of **granularity**. We can make one by allowing data items to be of various sizes and defining 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.

Each node in the tree can be locked individually. As we did in the two-phase locking protocol,we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode.

For example, if transaction *Ti* gets an **explicit lock** on file *Fc* of Figure 16.16, in exclusive mode, then it has an **implicit lock** in exclusive mode all the records belonging to that file. It does not need to lock the individual records of *Fc* explicitly.

Intention Lock Modes: If a node is locked in an intention mode, explicit locking is being done at a lower level of the tree (that is, at a finer granularity). Intention locks are put on all the ancestors of a node before that node is locked explicitly. Thus, a transaction does not need to search the entire tree to determine whether it can lock a node successfully. A transaction wishing to lock a node—say, *Q*—must traverse a path in the tree from the root to *Q*. While traversing the tree, the transaction locks the various nodes in an intention mode.

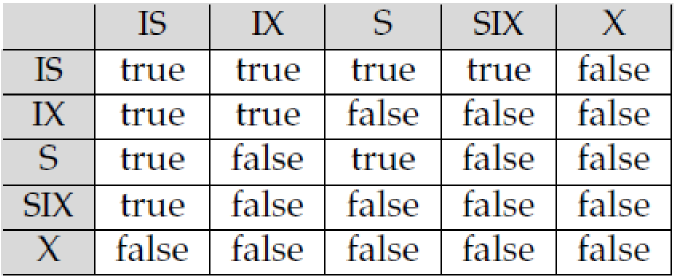
There is an intention mode associated with shared mode, and there is one with exclusive mode.

- If a node is locked in **intention-shared (IS) mode**, explicit locking is being done at a lower level of the tree, but with only shared-mode locks.

- If a node is locked in **intention-exclusive (IX) mode**, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks.

- If a node is locked in **shared and intention-exclusive (SIX) mode**, the subtree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

  The compatibility function for these lock modes is shown in Figure.



	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

Compatibility Matrix

The **multiple-granularity locking protocol**, which ensures serializability, is this:

Each transaction *Ti* can lock a node *Q* by following these rules:

1. It must observe the lock-compatibility function of Figure.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node *Q* in S or IS mode only if it currently has the parent of *Q* locked in either IX or IS mode.
4. It can lock a node *Q* in X, SIX, or IX mode only if it currently has the parent of *Q* locked in either IX or SIX mode.
5. It can lock a node only if it has not previously unlocked any node (that is, *Ti* is two phase).
6. It can unlock a node *Q* only if it currently has none of the children of *Q* locked.

Observe that the multiple-granularity protocol requires that locks be acquired in *top-down* (root-to-leaf) order, whereas locks must be released in *bottom-up* (leaf-to-root) order.

This protocol enhances concurrency and reduces lock overhead.

Deadlock is possible in this protocol, as it is in the two-phase locking protocol.

Multiversion Schemes

In **multiversion concurrency control** schemes, each write(*Q*) operation creates a new **version** of *Q*. When a transaction issues a read(*Q*) operation, the concurrency control manager selects one of the versions of *Q* to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability. It is also crucial, for performance reasons, that a transaction be able to determine easily and quickly which version of the data item should be read.

Multiversion Timestamp Ordering

With each transaction *Ti* in the system, we associate a unique static timestamp, denoted by TS(*Ti*). The database system assigns this timestamp before the transaction starts execution.

With each data item *Q*, a sequence of versions <*Q*1, *Q*2, . . .,*Qm*> is associated. Each version *Qk* contains three data fields:

- **Content** is the value of version *Qk*.
- **W-timestamp**(*Qk*) is the timestamp of the transaction that created version *Qk*.
- **R-timestamp**(*Qk*) is the largest timestamp of any transaction that successfully read version *Qk*.

A transaction—say, *Ti*—creates a new version *Qk* of data item *Q* by issuing a write(*Q*) operation. The content field of the version holds the value written by *Ti*. The system initializes the W-timestamp and R-timestamp to TS(*Ti*). It updates the R-timestamp value of *Qk* whenever a transaction *Tj* reads the content of *Qk*, and R-timestamp(*Qk*) < TS(*Tj*).

The **multiversion timestamp-ordering scheme** presented next ensures serializability. The scheme operates as follows. Suppose that transaction *Ti* issues a read(*Q*) or write(*Q*) operation. Let *Qk* denote the version of *Q* whose write timestamp is the largest write timestamp less than or equal to TS(*Ti*).

1. If transaction *Ti* issues a read(*Q*), then the value returned is the content of version *Qk*.
2. If transaction *Ti* issues write(*Q*), and if TS(*Ti*)<R-timestamp(*Qk*), then the system rolls back transaction *Ti*. On the other hand, if TS(*Ti*) = W-timestamp(*Qk*), the system overwrites the contents of *Qk*; otherwise it creates a new version of *Q*.

The justification for rule 1 is clear. A transaction reads the most recent version that comes before it in time. The second rule forces a transaction to abort if it is “too late” in doing a write. More precisely, if *Ti* attempts to write a version that some other transaction would have read, then we cannot allow that write to succeed.

Versions that are no longer needed are removed according to the following rule. Suppose that there are two versions, *Qk* and *Qj* , of a data item, and that both versions have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions *Qk* and *Qj* will not be used again, and can be deleted.

The multiversion timestamp-ordering scheme has the desirable property that a read request never fails and is never made to wait.

The scheme, however, suffers from two undesirable properties.

1. The reading of a data item also requires the updating of the R-timestamp field, resulting in two potential disk accesses, rather than one.

2. The conflicts between transactions are resolved through rollbacks, rather than through waits. This alternative may be expensive.

This multiversion timestamp-ordering scheme **does not ensure recoverability and cascadelessness.**

Multiversion Two-Phase Locking

The **multiversion two-phase locking protocol** attempts to combine the advantages of multiversion concurrency control with the advantages of two-phase locking. This protocol differentiates between **read-only transactions** and **update transactions**. Update transactions perform rigorous two-phase locking; that is, they hold all locks up to the end of the transaction. Thus, they can be serialized according to their commit order. Each version of a data item has a single timestamp. The timestamp in this case is not a real clock-based timestamp, but rather is a counter, which we will call the ts-counter, that is incremented during commit processing.

Read-only transactions are assigned a timestamp by reading the current value of ts-counter before they start execution; they follow the multiversion timestamp ordering protocol for performing reads. Thus, when a read-only transaction *Ti* issues a read(*Q*), the value returned is the contents of the version whose timestamp is the largest timestamp less than TS(*Ti*).

When an update transaction reads an item, it gets a shared lock on the item, and reads the latest version of that item. When an update transaction wants to write an item, it first gets an exclusive lock on the item, and then creates a new version of the data item. The write is performed on the new version, and the timestamp of the new version is initially set to a value ∞, a value greater than that of any possible timestamp.

When the update transaction *Ti* completes its actions, it carries out commit processing: First, *Ti* sets the timestamp on every version it has created to 1 more than the value of ts-counter; then, *Ti* increments ts-counter by 1. Only one update transaction is allowed to perform commit processing at a time.

As a result, read-only transactions that start after *Ti* increments ts-counter will see the values updated by *Ti*,whereas those that start before *Ti* increments ts-counter will see the value before the updates by *Ti*. In either case, read-only transactions never need to wait for locks.

Multiversion two-phase locking also ensures that schedules are recoverable and cascadeless.

Case Study of Oracle

Oracle's corporation, the world's leading enterprise company, provides enterprise software to the world's largest and most successful businesses. Oracle's information coordinator Group call center serves 7500 employees at Oracle's 2.2 million square foot headquarters facility in Redwood Shores, CA. The call center has improved its productivity using Amcom's automated speech recognition directory software and PC-attendant Console applications.

The Challenge:

The information coordinator group at Oracle corporation was searching for a way to increase the efficiency of the call center at Oracles's main headquarters in Redwood Shores, CA. The call center had become reliant on costly additional staffing to handle increasing call volumes. In addition, outdated phone consoles required operators to manually key extension numbers to transfer calls- a time consuming process prone to errors due to mistyped or transposed numbers.

The information coordinator group required a scalable, open standards Computer Telephony Integration (CTI) system that could bring a quick return on investment while raising call center performance. The system was required to run from an Oracle database and to provide both PC-attendant and automated speech recognition capabilities.

Oracle's Objective

- Reduce headcount of labour dedicated to call center activities.
- Process atleast 25% of all dial-zero calls through automated speech recognition.
- Reduce misdirected calls.
- Transition operators and callers quickly to using new system.
- Have access to robust statistical analysis to fine tune call center performance.

The Solution

The Oracle Information Coordinator Group chose a two-pronged solution from Amcom software consisting of smart-speech Automated Directory Application and smart console PC-attendant workstations.

The Smart Speech Directory provides total automation of directory assistance without operator intervention. All callers have to do is say the name of Oracle employee or department they are trying to search and the Smart Speech system automatically transfers the call.

The Amcom Smart Console PC attendant system, serves as an automation tool for Oracle's operators. Automated tool answers calls with each opeartor's pre-recorded greetings and announces extension numbers so that callers may dial directly in the future. A Directory service enables operators to quickly look up extensions and transfers calls with a few keystrokes.

The Result

- The Oracle Information Coordinator Group was able to eliminate three staff positions as a result of implementing Amcom system.
- The Oracle headquarters call center is now answering 98% of all calls within 10 seconds.
- Operators have quick access to up to date information and a complete set of information- rich, time saving tools.
- Operator transfers are seldom misdirected, providing greater satisfaction to callers.
- Callers dialing the Smart Speech Directory line are connected immediately, without having to wait for an operator.

The Phantom Phenomenon

Consider transaction T_{29} that executes the following SQL query on the bank database:

select sum(*balance*)
from *account*
where *branch-name* = 'Perryridge'

Transaction T_{29} requires access to all tuples of the *account* relation pertaining to the Perryridge branch.

Let T_{30} be a transaction that executes the following SQL insertion:

insert into *account*
values (A-201, 'Perryridge', 900)

Let S be a schedule involving T_{29} and T_{30}.
In the phantom problem, a transaction accesses a relation more than once with the same predicate in the same transaction, but sees new phantom tuples on re-access that were not seen on the first access.

In other words, a transaction reruns a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

In effect, T_{29} and T_{30} conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. This problem is called the phantom phenomenon.