Unit-5

**Concurrency Control** in Database Management System is a procedure of managing simultaneous operations without conflicting with each other. It ensures that Database transactions are performed concurrently and accurately to produce correct results without violating data integrity of the respective Database.

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, it would have a mix of READ and WRITE operations and hence the concurrency is a challenge.

DBMS Concurrency Control is used to address such conflicts, which mostly occur with a multi-user system. Therefore, Concurrency Control is the most important element for proper functioning of a Database Management System where two or more database transactions are executed simultaneously, which require access to the same data.

Potential problems of Concurrency

Here, are some issues which you will likely to face while using the DBMS Concurrency Control method:

* Lost Updates occur when multiple transactions select the same row and update the row based on the value selected
* Uncommitted dependency issues occur when the second transaction selects a row which is updated by another transaction (dirty read)
* Non-Repeatable Read occurs when a second transaction is trying to access the same row several times and reads different data each time.
* Incorrect Summary issue occurs when one transaction takes summary over the value of all the instances of a repeated data-item, and second transaction update few instances of that specific data-item. In that situation, the resulting summary does not reflect a correct result.

Why use Concurrency method?

Reasons for using Concurrency control method is DBMS:

* To apply Isolation through mutual exclusion between conflicting transactions
* To resolve read-write and write-write conflict issues
* To preserve database consistency through constantly preserving execution obstructions
* The system needs to control the interaction among the concurrent transactions. This control is achieved using concurrent-control schemes.
* Concurrency control helps to ensure serializability

Example

Assume that two people who go to electronic kiosks at the same time to buy a movie ticket for the same movie and the same show time.

However, there is only one seat left in for the movie show in that particular theatre. Without concurrency control in DBMS, it is possible that both moviegoers will end up purchasing a ticket. However, concurrency control method does not allow this to happen. Both moviegoers can still access information written in the movie seating database. But concurrency control only provides a ticket to the buyer who has completed the transaction process first.

**Concurrency Control Protocols**

Different concurrency control protocols offer different benefits between the amount of concurrency they allow and the amount of overhead that they impose. Following are the Concurrency Control techniques in DBMS:

* Lock-Based Protocols
* Two Phase Locking Protocol
* Timestamp-Based Protocols
* Multi-version Concurrency control
* Validation-Based Protocols

**Lock-based Protocols**

Lock Based Protocols in DBMS is a mechanism in which a transaction cannot Read or Write the data until it acquires an appropriate lock. Lock based protocols help to eliminate the concurrency problem in DBMS for simultaneous transactions by locking or isolating a particular transaction to a single user.

A lock is a data variable which is associated with a data item. This lock signifies that operations that can be performed on the data item. Locks in DBMS help synchronize access to the database items by concurrent transactions.

All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.

Binary Locks: A Binary lock on a data item can either locked or unlocked states.

Shared/exclusive: This type of locking mechanism separates the locks in DBMS based on their uses. If a lock is acquired on a data item to perform a write operation, it is called an exclusive lock.

1. Shared Lock (S):

A shared lock is also called a Read-only lock. With the shared lock, the data item can be shared between transactions. This is because you will never have permission to update data on the data item.

For example, consider a case where two transactions are reading the account balance of a person. The database will let them read by placing a shared lock. However, if another transaction wants to update that account’s balance, shared lock prevent it until the reading process is over.

2. Exclusive Lock (X):

With the Exclusive Lock, a data item can be read as well as written. This is exclusive and can’t be held concurrently on the same data item. X-lock is requested using lock-x instruction. Transactions may unlock the data item after finishing the ‘write’ operation.

For example, when a transaction needs to update the account balance of a person. You can allows this transaction by placing X lock on it. Therefore, when the second transaction wants to read or write, exclusive lock prevent this operation.

3. Simplistic Lock Protocol

This type of lock-based protocols allows transactions to obtain a lock on every object before beginning operation. Transactions may unlock the data item after finishing the ‘write’ operation.

4. Pre-claiming Locking

Pre-claiming lock protocol helps to evaluate operations and create a list of required data items which are needed to initiate an execution process. In the situation when all locks are granted, the transaction executes. After that, all locks release when all of its operations are over.

Starvation

Starvation is the situation when a transaction needs to wait for an indefinite period to acquire a lock.

Following are the reasons for Starvation:

* When waiting scheme for locked items is not properly managed
* In the case of resource leak
* The same transaction is selected as a victim repeatedly

Deadlock

Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.

**Two-Phase Locking Techniques for Concurrency Control**

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the data-base. Locks are used as a means of synchronizing the access by concurrent transactions to the database items. In Section 22.1.1 we discuss the nature and types of locks. Then, in Section 22.1.2 we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 22.1.3 we describe two problems associated with the use of locks—deadlock and starvation—and show how these problems are handled in concurrency control protocols.

**1. Types of Locks and System Lock Tables**

Several types of locks are used in concurrency control. To introduce locking concepts gradually, first we discuss binary locks, which are simple, but are also *too* *restrictive for database concurrency control purposes*, and so are not used in practice.Then we discuss *shared/exclusive* locks—also known as *read/write* locks—which provide more general locking capabilities and are used in practical database locking schemes. In Section 22.3.2 we describe an additional type of lock called a *certify lock*, and show how it can be used to improve performance of locking protocols.

**Binary Locks.**A**binary lock**can have two**states**or**values:**locked and unlocked (or1 and 0, for simplicity). A distinct lock is associated with each database item *X*. If the value of the lock on *X* is 1, item *X cannot be accessed* by a database operation that requests the item. If the value of the lock on *X* is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item *X* as **lock**(***X***).

Two operations, lock\_item and unlock\_item, are used with binary locking. A transaction requests access to an item *X* by first issuing a **lock\_item(*X*)** operation. If LOCK(*X*) = 1, the transaction is forced to wait. If LOCK(*X*) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item *X*. When the transaction is through using the item, it issues an **unlock\_item(*X*)** operation, which sets LOCK(*X*) back to 0 (**unlocks** the item) so that *X* may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock\_item(*X*) and unlock\_item(*X*) operations is shown in Figure 22.1.

lock\_item(X):

B:   if LOCK(X) = 0 (\* item is unlocked \*)

then LOCK(X) ←1 (\* lock the item \*)

else

begin

wait (until LOCK(X) = 0

and the lock manager wakes up the transaction);

go to B

end;

unlock\_item(X):

LOCK(X) ← 0; (\* unlock the item \*)

if any transactions are waiting

then wakeup one of the waiting transactions;

Figure 22.1 Lock and unlock operations for binary locks.

Notice that the lock\_item and unlock\_item operations must be implemented as indivisible units (known as **critical sections** in operating systems); that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 22.1, the wait command within the lock\_item(*X*) operation is usually implemented by putting the transaction in a waiting queue for item *X* until *X* is unlocked and the transaction can be granted access to it. Other transactions that also want to access *X* are placed in the same queue. Hence, the wait command is considered to be outside the lock\_item operation.

 It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item *X* in the database. In its simplest form, each lock can be a record with three fields: <Data\_item\_name, LOCK, Locking\_transaction> plus a queue for transactions that are waiting to access the item. The system needs to maintain *only these records for the items that are currently locked* in a **lock table**, which could be organized as a hash file on the item name. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager sub-system**to keep track of and control access to locks.

 If the simple binary locking scheme described here is used, every transaction must obey the following rules:

* A transaction *T* must issue the operation lock\_item(*X*) before any read\_item(*X*) or write\_item(*X*) operations are performed in *T.*
* A transaction *T* must issue the operation unlock\_item(*X*) after all read\_item(*X*) and write\_item(*X*) operations are completed in *T.*
* A transaction *T* will not issue a lock\_item(*X*) operation if it already holds the lock on item *X*.1
* A transaction *T* will not issue an unlock\_item(*X*) operation unless it already holds the lock on item *X*.

These rules can be enforced by the lock manager module of the DBMS. Between the lock\_item(*X*) and unlock\_item(*X*) operations in transaction *T*, *T* is said to **hold the** **lock**on item*X*. At most one transaction can hold the lock on a particular item.Thus no two transactions can access the same item concurrently.

Shared/Exclusive (or Read/Write) Locks. The preceding binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item *X* if they all access *X* for *reading purposes only*. This is because read operations on the same item by different transactions are not conflicting (see Section 21.4.1). However, if a transaction is to write an item *X*, it must have exclusive access to *X*. For this purpose, a different type of lock called a **multiple-mode lock** is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read\_lock(*X*), write\_lock(*X*), and unlock(*X*). A lock associated with an item *X*, LOCK(*X*), now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A**read-locked item**is also called**share-locked**because other transactionsare allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

 One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table. Each record in the lock table will have four fields: <Data\_item\_name, LOCK, No\_of\_reads, Locking\_transaction(s)>. Again, to save space, the system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If LOCK(*X*)=write-locked, the value of locking\_transaction(s) is a single transaction that holds the exclusive (write) lock on *X*. If LOCK(*X*)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on *X*. The three operations read\_lock(*X*), write\_lock(*X*), and unlock(*X*) are described in Figure 22.2. As before, each of the three locking operations should be considered indivisible; no interleav-ing should be allowed once one of the operations is started until either the opera-tion terminates by granting the lock or the transaction is placed in a waiting queue for the item.

 When we use the shared/exclusive locking scheme, the system must enforce the fol-lowing rules:

         A transaction *T* must issue the operation read\_lock(*X*) or write\_lock(*X*) before any read\_item(*X*) operation is performed in *T.*

            A transaction *T* must issue the operation write\_lock(*X*) before any write\_item(*X*) operation is performed in *T.*

read\_lock(*X*):

         if LOCK(*X*) = “unlocked”

 then begin LOCK(*X*) ← “read-locked”;

no\_of\_reads(*X*) ← 1

 end

 else if LOCK(*X*) = “read-locked”

 then no\_of\_reads(*X*) ← no\_of\_reads(*X*) + 1 else begin

 wait (until LOCK(*X*) = “unlocked”

and the lock manager wakes up the transaction); go to B

 end; write\_lock(*X*):

        if LOCK(*X*) = “unlocked”

 then LOCK(*X*) ← “write-locked” else begin

 wait (until LOCK(*X*) = “unlocked”

 and the lock manager wakes up the transaction); go to B

             end;

            unlock (X):

            if LOCK(X) = “write-locked”

            then begin LOCK(X) ← “unlocked”;

            wakeup one of the waiting transactions, if any

            end

            else it LOCK(X) = “read-locked”

            then begin

            no\_of\_reads(X) ← no\_of\_reads(X) −1;

            if no\_of\_reads(X) = 0

                                      then begin LOCK(X) = “unlocked”;

            wakeup one of the waiting transactions, if any

            end                     shared-exclusive)

            end;

                  Figure : Locking and unlocking operations for two- mode (read-write or locks.

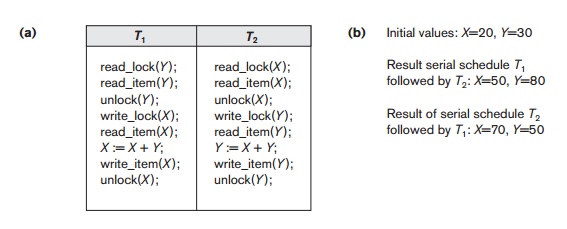
* A transaction *T* must issue the operation unlock(*X*) after all read\_item(*X*) and write\_item(*X*) operations are completed in *T.*3
* A transaction *T* will not issue a read\_lock(*X*) operation if it already holds a read (shared) lock or a write (exclusive) lock on item *X*. This rule may be relaxed, as we discuss shortly.
* A transaction *T* will not issue a write\_lock(*X*) operation if it already holds a read (shared) lock or write (exclusive) lock on item *X*. This rule may also be relaxed, as we discuss shortly.
* A transaction *T* will not issue an unlock(*X*) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

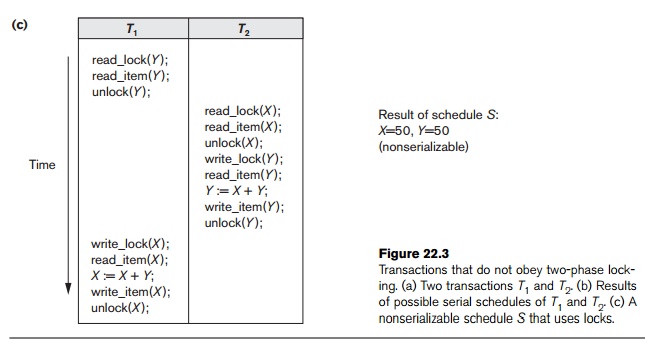
Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item *X* is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction *T* to issue a read\_lock(*X*) and then later to **upgrade** the lock by issuing a write\_lock(*X*) operation. If *T* is the only transaction holding a read lock on *X* at the time it issues the write\_lock(*X*) operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction *T* to issue a write\_lock(*X*) and then later to **downgrade**the lock by issuing aread\_lock(*X*) operation. When upgrading anddowngrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking\_transaction(s) field) to store the information on which transactions hold locks on the item. The descriptions of the read\_lock(*X*) and write\_lock(*X*) operations in Figure 22.2 must be changed appropri-ately to allow for lock upgrading and downgrading. We leave this as an exercise for the reader.

 Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 22.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 22.3(a) the items *Y* in *T*1 and *X* in *T*2 were unlocked too early. This allows a schedule such as the one shown in Figure 22.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction. The best-known protocol, two-phase locking, is described in the next section.

**2. Guaranteeing Serializability by Two-Phase Locking**

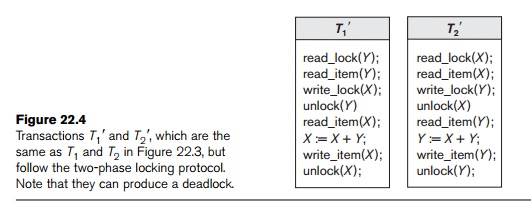
 A transaction is said to follow the **two-phase locking protocol** if *all* locking operations (read\_lock, write\_lock) precede the *first* unlock operation in the transaction. Such a transaction can be divided into two phases: an **expanding** or **growing (first)** **phase**, during which new locks on items can be acquired but none can be released;and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the





shrinking phase. Hence, a read\_lock(*X*) operation that downgrades an already held write lock on *X* can appear only in the shrinking phase.

Transactions *T*1 and *T*2 in Figure 22.3(a) do not follow the two-phase locking protocol because the write\_lock(*X*) operation follows the unlock(*Y*) operation in *T*1, and similarly the write\_lock(*Y*) operation follows the unlock(*X*) operation in *T*2. If we enforce two-phase locking, the transactions can be rewritten as *T*1 and *T*2 , as shown in Figure 22.4. Now, the schedule shown in Figure 22.3(c) is not permitted for *T*1 and *T*2 (with their modified order of locking and unlocking operations) under the rules of locking described in Section 22.1.1 because *T*1 will issue its write\_lock(*X*) *before* it unlocks item *Y*; consequently, when *T*2 issues its read\_lock(*X*), it is forced to wait until *T*1 releases the lock by issuing an unlock (*X*) in the schedule.



It can be proved that, if *every* transaction in a schedule follows the two-phase locking protocol, the schedule is *guaranteed to be serializable*, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase locking rules, also enforces serializability.

 Two-phase locking may limit the amount of concurrency that can occur in a schedule because a transaction *T* may not be able to release an item *X* after it is through using it if *T* must lock an additional item *Y* later; or conversely, *T* must lock the additional item *Y* before it needs it so that it can release *X*. Hence, *X* must remain locked by *T* until all items that the transaction needs to read or write have been locked; only then can *X* be released by *T*. Meanwhile, another transaction seeking to access *X* may be forced to wait, even though *T* is done with *X*; conversely, if *Y* is locked earlier than it is needed, another transaction seeking to access *Y* is forced to wait even though *T* is not using *Y* yet. This is the price for guaranteeing serializability of all schedules without having to check the schedules themselves.

 Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit *all possible* serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

 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*. Recall from Section 21.1.2 thatthe **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 neededcannot 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, as we will see in Section 22.1.3 when we discuss the deadlock problem. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in many situations.

 In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules (see Section 21.4). In this variation, a transaction *T* does not release any of its exclusive (write) locks until *after* it commits or aborts. Hence, no other transaction can read or write an item that is written by *T* unless *T* has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free. A more restrictive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction *T* does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL. Notice the difference between conservative and rigorous 2PL: the former must lock all its items *before it starts*, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until *after it terminates* (by committing or aborting), so the transaction is in its expanding phase until it ends.

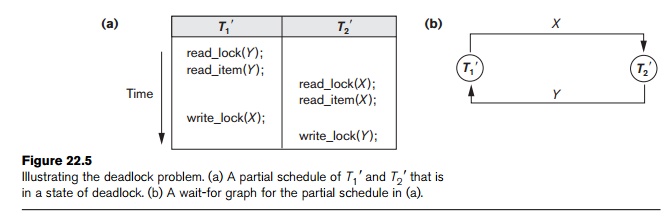
 In many cases, the **concurrency control subsystem** itself is responsible for generating the read\_lock and write\_lock requests. For example, suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction *T* issues a read\_item(*X*), the system calls the read\_lock(*X*) operation on behalf of *T*. If the state of LOCK(*X*) is write\_locked by some other transaction *T* , the system places *T* in the waiting queue for item *X*; otherwise, it grants the read\_lock(*X*) request and permits the read\_item(*X*) operation of *T* to execute. On the other hand, if transaction *T* issues a write\_item(*X*), the system calls the write\_lock(*X*) operation on behalf of *T*. If the state of LOCK(*X*) is write\_locked or read\_locked by some other transaction *T* , the system places *T* in the waiting queue for item *X*; if the state of LOCK(*X*) is read\_locked and *T*itself is the only transaction holding the read lock on*X*, the system upgrades thelock to write\_locked and permits the write\_item(*X*) operation by *T*. Finally, if the state of LOCK(*X*) is unlocked, the system grants the write\_lock(*X*) request and permits the write\_item(*X*) operation to execute. After each action, the system must update its lock table appropriately.

 The use of locks can cause two additional problems: deadlock and starvation. We discuss these problems and their solutions in the next section.

**3. Dealing with Deadlock and Starvation**

**Deadlock**occurs when*each*transaction*T*in a set of*two or more transactions*iswaiting for some item that is locked by some other transaction *T* in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other trans-actions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock. A simple example is shown in Figure 22.5(a), where the two transactions *T*1 and *T*2 are deadlocked in a partial schedule; *T*1is in the waiting queue for*X*, which is locked by*T*2, while*T*2is in the waitingqueue for *Y*, which is locked by *T*1 . Meanwhile, neither *T*1 nor *T*2 nor any other transaction can access items *X* and *Y*.

Deadlock Prevention Protocols. One way to prevent deadlock is to use a **deadlock prevention protocol**.One deadlock prevention protocol, which is used



in conservative two-phase locking, requires that every transaction lock *all the items* *it needs in advance*(which is generally not a practical assumption)—if any of theitems cannot be obtained, none of the items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously this solution further limits concurrency. A second protocol, which also limits concurrency, involves *ordering all the items*in the database and making sure that a transaction that needsseveral items will lock them according to that order. This requires that the program-mer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

 A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situation: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction? Some of these techniques use the concept of **transaction timestamp** TS(*T*), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which trans-actions are started; hence, if transaction *T*1 starts before transaction *T*2, then TS(*T*1) < TS(*T*2). Notice that the *older* transaction (which starts first) has the *smaller* time-stamp value. Two schemes that prevent deadlock are called *wait-die* and *wound-wait*. Suppose that transaction*Ti*tries to lock an item*X*but is not able to because*X*is locked by some other transaction *Tj* with a conflicting lock. The rules followed by these schemes are:

**Wait-die.**IfTS(*Ti*) <TS(*Tj*), then (*Ti*older than*Tj*)*Ti*is allowed to wait;otherwise (*Ti* younger than *Tj*) abort *Ti* (*Ti* *dies*) and restart it later *with the* *same timestamp.*

**Wound-wait.**IfTS(*Ti*) <TS(*Tj*), then (*Ti*older than*Tj*) abort*Tj*(*Ti**wounds**Tj*) and restart it later*with the same timestamp;*otherwise (*Ti*younger than*Tj*)*Ti*is allowed to wait.

 In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it. Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing. It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created. However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may *never actually cause a* *deadlock*.

 Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the **no** **waiting algorithm**, if a transaction is unable to obtain a lock, it is immediatelyaborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly. The **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction *Ti* tries to lock an item *X*but is not able to do so because*X*is locked by some other transaction*Tj*with aconflicting lock. The cautious waiting rules are as follows:

**Cautious waiting.**If*Tj*is not blocked (not waiting for some other lockeditem), then *Ti* is blocked and allowed to wait; otherwise abort *Ti.*

 It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time *b*(*T*) at which each blocked transaction *T* was blocked, if the two transactions *Ti* and *Tj* above both become blocked, and *Ti* is waiting for *Tj*, then *b*(*Ti*) < *b*(*Tj*), since *Ti* can only wait for *Tj* at a time when *T* *j* is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

 Deadlock Detection. A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light. On the other hand, if transactions are long and each transaction uses many items, or if the transaction load is quite heavy, it may be advantageous to use a deadlock prevention scheme.

 A simple way to detect a state of deadlock is for the system to construct and maintain a **wait-for graph**. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction *Ti* is waiting to lock an item *X* that is currently locked by a transaction *Tj*, a directed edge (*Ti* → *Tj*) is created in the wait-for graph. When *Tj* releases the lock(s) on the items that *Ti* was waiting for, the directed edge is dropped from the wait-for graph. We have a state of dead-lock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining *when* the system should check for a deadlock. One possibility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead. Criteria such as the number of currently executing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 22.5(a).

 If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as **victim selection**. The algorithm for victim selection should generally avoid selecting transactions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

 Timeouts. Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

 Starvation. Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served**queue; transactions are enabled to lock an item in the orderin which they originally requested the lock. Another scheme allows some transactions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

**Concurrency Control Based on Timestamp Ordering**

The use of locks, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem. A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equivalent serial schedule. In Section 22.2.1 we discuss timestamps, and in Section 22.2.2 we discuss how serializability is enforced by ordering transactions based on their timestamps.

**1. Timestamps**

 Recall that a **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction*T*as**TS(*T*)**.Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

 Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction time-stamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite max-imum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

**2. The Timestamp Ordering Algorithm**

 The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only* *equivalent serial schedule permitted*has the transactions in order of their timestampvalues. This is called **timestamp ordering (TO**). Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction time-stamps. The algorithm must ensure that, for each item accessed by *conflicting operations*in the schedule, the order in which the item is accessed does not violate thetimestamp order. To do this, the algorithm associates with each database item *X* two timestamp (**TS**) values:

**read\_TS(*X*).**The**read timestamp**of item*X*is the largest timestamp amongall the timestamps of transactions that have successfully read item *X*—that is, read\_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has read *X* successfully.

        **write\_TS(*X*).**The**write timestamp**of item*X*is the largest of all the time-

stamps of transactions that have successfully written item *X*—that is, write\_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has written *X* successfully.

 Basic Timestamp Ordering (TO). Whenever some transaction *T* tries to issue a read\_item(*X*) or a write\_item(*X*) operation, the **basic TO** algorithm compares the timestamp of *T* with read\_TS(*X*) and write\_TS(*X*) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transac-tion *T* is aborted and resubmitted to the system as a new transaction with a *new* *timestamp*. If*T*is aborted and rolled back, any transaction*T*1that may have used avalue written by *T* must also be rolled back. Similarly, any transaction *T*2 that may have used a value written by *T*1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. An *additional* *protocol*must be enforced to ensure that the schedules are recoverable, cascadeless,or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp order-ing in the following two cases:

         Whenever a transaction *T* issues a write\_item(*X*) operation, the following is checked:

        If read\_TS(*X*) > TS(*T*) or if write\_TS(*X*) > TS(*T*), then abort and roll back *T*and reject the operation. This should be done because some*younger*transaction with a timestamp greater than TS(*T*)—and hence *after T* in the timestamp ordering—has already read or written the value of item *X* before *T* had a chance to write *X*, thus violating the timestamp ordering.

 ·         If the condition in part (a) does not occur, then execute the write\_item(*X*) operation of *T* and set write\_TS(*X*) to TS(*T*).

* Whenever a transaction *T* issues a read\_item(*X*) operation, the following is checked:
* If write\_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the opera-tion. This should be done because some younger transaction with time-stamp greater than TS(*T*)—and hence *after T* in the timestamp ordering—has already written the value of item *X* before *T* had a chance to read *X*.
* If write\_TS(*X*) ≤ TS(*T*), then execute the read\_item(*X*) operation of *T* and set read\_TS(*X*) to the *larger* of TS(*T*) and the current read\_TS(*X*).

 Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*, like the 2PL protocol. However, some schedules are possibleunder each protocol that are not allowed under the other. Thus, *neither* protocol allows *all possible* serializable schedules. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO). A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable. In this variation, a transaction *T* that issues a read\_item(*X*) or write\_item(*X*) such that TS(*T*) > write\_TS(*X*) has its read or write operation *delayed* until the transaction *T* that *wrote* the value of *X* (hence TS(*T* ) = write\_TS(*X*)) has committed or aborted. To implement this algorithm, it is necessary to simulate the locking of an item *X* that has been written by transaction *T* until *T* is either com-mitted or aborted. This algorithm *does not cause deadlock*, since *T* waits for *T* only if TS(*T*) > TS(*T* ).

 Thomas’s Write Rule. A modification of the basic TO algorithm, known as **Thomas’s write rule**, does not enforce conflict serializability, but it rejects fewerwrite operations by modifying the checks for the write\_item(*X*) operation as follows:

         If read\_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation.

        If write\_TS(*X*) > TS(*T*), then do not execute the write operation but continue

 processing. This is because some transaction with timestamp greater than TS(*T*)—and hence after *T* in the timestamp ordering—has already written the value of *X*. Thus, we must ignore the write\_item(*X*) operation of *T* because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).

         If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write\_item(*X*) operation of *T* and set write\_TS(*X*) to TS(*T*).

**Multiversion Concurrency Control Techniques**

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control**, because several versions (values) of an item are maintained. When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older* *version*of the item to maintain serializability. When a transaction writes an item, itwrites a *new version* and the old version(s) of the item are retained. Some multiver-sion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a *temporal database* (see Secton 26.2), which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

Several multiversion concurrency control schemes have been proposed. We discuss two schemes here, one based on timestamp ordering and the other based on 2PL. In addition, the validation concurrency control method (see Section 22.4) also maintains multiple versions.

**1. Multiversion Technique Based on Timestamp Ordering**

In this method, several versions *X*1, *X*2, ..., *Xk* of each data item *X* are maintained. For *each version*, the value of version *Xi* and the following two timestamps are kept:

**read\_TS(*Xi*).**The**read timestamp**of*Xi*is the largest of all the timestamps oftransactions that have successfully read version *Xi*.

**write\_TS(*Xi*).**The**write timestamp**of*Xi*is the timestamp of the transaction that wrote the value of version *Xi*.

 Whenever a transaction *T* is allowed to execute a write\_item(*X*) operation, a new version *Xk*+1 of item *X* is created, with both the write\_TS(*Xk*+1) and the read\_TS(*Xk*+1) set to TS(*T*). Correspondingly, when a transaction *T* is allowed to read the value of version *Xi*, the value of read\_TS(*Xi*) is set to the larger of the current read\_TS(*Xi*) and TS(*T*).

 To ensure serializability, the following rules are used:

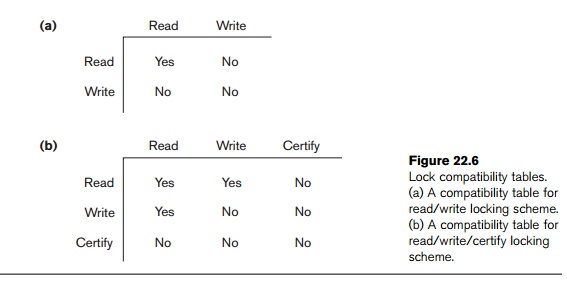
         If transaction *T* issues a write\_item(*X*) operation, and version *i* of *X* has the highest write\_TS(*Xi*) of all versions of *X* that is also *less than or equal to* TS(*T*), and read\_TS(*Xi*) > TS(*T*), then abort and roll back transaction *T*; otherwise, create a new version *Xj* of *X* with read\_TS(*Xj*) = write\_TS(*Xj*) = TS(*T*).

        If transaction *T* issues a read\_item(*X*) operation, find the version *i* of *X* that has the highest write\_TS(*Xi*) of all versions of *X* that is also *less than or equal* *to*TS(*T*); then return the value of*Xi*to transaction*T,*and set the value ofread\_TS(*Xi*) to the larger of TS(*T*) and the current read\_TS(*Xi*).

 As we can see in case 2, a read\_item(*X*) is always successful, since it finds the appropriate version *Xi* to read based on the write\_TS of the various existing versions of *X*. In case 1, however, transaction *T* may be aborted and rolled back. This happens if *T* attempts to write a version of *X* that should have been read by another transaction *T*whose timestamp isread\_TS(*Xi*); however,*T*has already read version*Xi*, whichwas written by the transaction with timestamp equal to write\_TS(*Xi*). If this conflict occurs, *T* is rolled back; otherwise, a new version of *X*, written by transaction *T*, is created. Notice that if *T* is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction *T* should not be allowed to commit until after all the transactions that have written some version that *T* has read have committed.

**2. Multiversion Two-Phase Locking Using Certify Locks**

 In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and *certify*, instead of just the two modes (read, write) discussed previously. Hence, the state of LOCK(*X*) for an item *X* can be one of read-locked, write-locked, certify-locked, or unlocked. In the standard locking scheme, with only read and write locks (see Section 22.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a). An entry of *Yes* means that if a transaction *T* holds the type of lock specified in the column header



on item *X* and if transaction *T* requests the type of lock specified in the row header on the same item *X*, then *T can obtain the lock* because the locking modes are compatible. On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so *T must wait* until *T releases* the lock.

 In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions *T* to read an item *X* while a single transaction *T* holds a write lock on *X*. This is accomplished by allowing *two versions* for each item *X*; one version must always have been written by some committed transaction. The second version *X* is created when a transaction *T* acquires a write lock on the item. Other transactions can continue to read the *committed version* of *X* while *T* holds the write lock. Transaction *T* can write the value of *X* as needed, without affecting the value of the committed version *X*. However, once *T* is ready to commit, it must obtain a **certify lock**on all items that it currently holds write locks on before it can commit.The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transactions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version *X* of the data item is set to the value of version *X* , version *X* is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 22.6(b).

 In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive certify locks on *all the items* it has updated. It can be shown that this scheme avoids cascading aborts, since transactions are only allowed to read the version *X* that was written by a committed transaction. However, deadlocks may occur if upgrading of a read lock to a write lock is allowed, and these must be handled by variations of the techniques discussed in Section 22.1.3.

**Validation (Optimistic) Concurrency Control Techniques**

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done *before* a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transaction execution, with the effect of slowing down the transactions.

In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**,*no checking*is done while the transaction is executing.Several theoretical concurrency control methods are based on the validation technique. We will describe only one scheme here. In this scheme, updates in the trans-action are *not* applied directly to the database items until the transaction reaches its end. During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction.6 At the end of transaction execution, a **validation phase**checks whether any of the transaction’s updates violate serializability. Certain information needed by the validation phase must be kept by the system. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

        **Read phase.**A transaction can read values of committed data items from thedatabase. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.

        **Validation phase.**Checking is performed to ensure that serializability willnot be violated if the transaction updates are applied to the database.

**Write phase.**If the validation phase is successful, the transaction updates areapplied to the database; otherwise, the updates are discarded and the trans-action is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the validation phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later. Under these circumstances, optimistic techniques do not work well. The techniques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.

The optimistic protocol we describe uses transaction timestamps and also requires that the write\_sets and read\_sets of the transactions be kept by the system. Additionally, *start* and *end* times for some of the three phases need to be kept for each transaction. Recall that the write\_set of a transaction is the set of items it writes, and the read\_set is the set of items it reads. In the validation phase for transaction *Ti*, the protocol checks that *Ti* does not interfere with any committed transactions or with any other transactions currently in their validation phase. The validation phase for *Ti* checks that, for *each* such transaction *Tj* that is either committed or is in its validation phase, *one* of the following conditions holds:

         Transaction *Tj* completes its write phase before *Ti* starts its read phase.

*Ti*starts its write phase after*Tj*completes its write phase, and theread\_setof *Ti* has no items in common with the write\_set of *Tj*.

        Both the read\_set and write\_set of *Ti* have no items in common with the write\_set of *Tj*, and *Tj* completes its read phase before *Ti* completes its read phase.

 When validating transaction *Ti*, the first condition is checked first for each transaction *Tj*, since (1) is the simplest condition to check. Only if condition 1 is false is condition 2 checked, and only if (2) is false is condition 3—the most complex to evaluate—checked. If any one of these three conditions holds, there is no interference and *Ti* is validated successfully. If *none* of these three conditions holds, the validation of transaction *Ti* fails and it is aborted and restarted later because interference *may* have occurred.

# Recoverability in DBMS

A transaction may not execute completely due to hardware failure, system crash or software issues. In that case, we have to roll back the failed transaction. But some other transaction may also have used values produced by the failed transaction. So we have to roll back those transactions as well.

**Recoverable Schedules:**

* Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.  
  **Example 1:**

S1: R1(x), **W1(x)**, R2(x), R1(y), R2(y),

**W2(x)**, W1(y), **C1**, **C2**;

Given schedule follows order of **Ti->Tj => C1->C2**. Transaction T1 is executed before T2 hence there is no chances of conflict occur. R1(x) appears before W1(x) and transaction T1 is committed before T2 i.e. completion of first transaction performed first update on data item x, hence given schedule is recoverable.

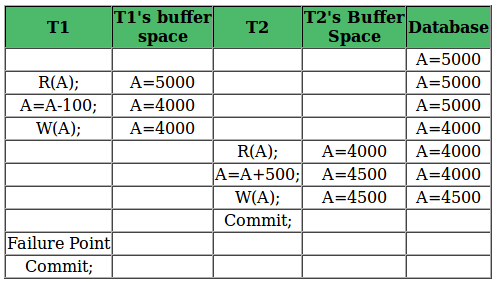
**Example 2:** Consider the following schedule involving two transactions T1 and T2.

| T1 | T2 |
| --- | --- |
| R(A) |  |
| W(A) |  |
|  | W(A) |
|  | R(A) |
| commit |  |
|  | commit |

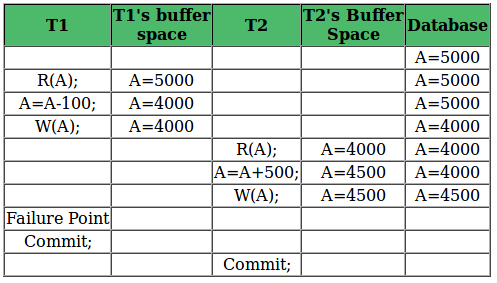
This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

**Irrecoverable Schedule:**

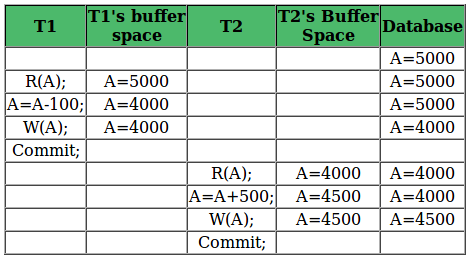
* The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. T2 commits. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. But we have already committed that. So this schedule is irrecoverable schedule. When Tj is reading the value updated by Ti and Tj is committed before committing of Ti, the schedule will be irrecoverable.



**Recoverable with Cascading Rollback:**

* The table below shows a schedule with two transactions, T1 reads and writes A and that value is read and written by T2. But later on, T1 fails. So we have to rollback T1. Since T2 has read the value written by T1, it should also be rollbacked. As it has not committed, we can rollback T2 as well. So it is recoverable with cascading rollback. Therefore, if Tj is reading value updated by Ti and commit of Tj is delayed till commit of Ti, the schedule is called recoverable with cascading rollback.  
  [](https://media.geeksforgeeks.org/wp-content/uploads/schedule5.png)

**Cascadeless Recoverable Rollback:**

* The table below shows a schedule with two transactions, T1 reads and writes A and commits and that value is read by T2. But if T1 fails before commit, no other transaction has read its value, so there is no need to rollback other transaction. So this is a Cascadeless recoverable schedule. So, if Tj reads value updated by Ti only after Ti is committed, the schedule will be cascadeless recoverable.  
  [](https://media.geeksforgeeks.org/wp-content/uploads/schedult3.png)

SERILIAZABULITY

A schedule is serialized if it is equivalent to a serial schedule. A concurrent schedule must ensure it is the same as if executed serially means one after another. It refers to the sequence of actions such as read, write, abort, commit are performed in a serial manner.

## Example

Let’s take two transactions T1 and T2,

If both transactions are performed without interfering each other then it is called as serial schedule, it can be represented as follows −

| **T1** | **T2** |
| --- | --- |
| READ1(A) |  |
| WRITE1(A) |  |
| READ1(B) |  |
| C1 |  |
|  | READ2(B) |
|  | WRITE2(B) |
|  | READ2(B) |
|  | C2 |

**Non serial schedule** − When a transaction is overlapped between the transaction T1 and T2.

## Example

Consider the following example −

| **T1** | **T2** |
| --- | --- |
| READ1(A) |  |
| WRITE1(A) |  |
|  | READ2(B) |
|  | WRITE2(B) |
| READ1(B) |  |
| WRITE1(B) |  |
| READ1(B) |  |

## Types of serializability

There are two types of serializability −

## View serializability

A schedule is view-serializability if it is viewed equivalent to a serial schedule.

The rules it follows are as follows −

T1 is reading the initial value of A, then T2 also reads the initial value of A.

T1 is the reading value written by T2, then T2 also reads the value written by T1.

T1 is writing the final value, and then T2 also has the write operation as the final value.

## Conflict serializability

It orders any conflicting operations in the same way as some serial execution. A pair of operations is said to conflict if they operate on the same data item and one of them is a write operation.

That means

Readi(x) readj(x) - non conflict  read-read operation

Readi(x) writej(x) - conflict       read-write operation.

Writei(x) readj(x) - conflict       write-read operation.

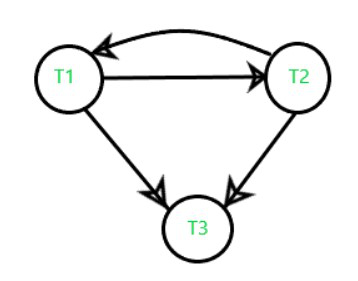
Writei(x) writej(x) - conflict      write-write operation.

# View Serializability in DBMS

**Example :**Understanding View-Serializability first with a **Schedule S1 :**

| T1 | T2 | T3 |
| --- | --- | --- |
| a=100  **read(a)** |  |  |
|  | a=a-40  **write(a) //60** |  |
| a=a-40  **write(a) //20** |  |  |
|  |  | a=a-20  **write(a) //0** |

So, its Conflict Precedence Graph is as follows – 

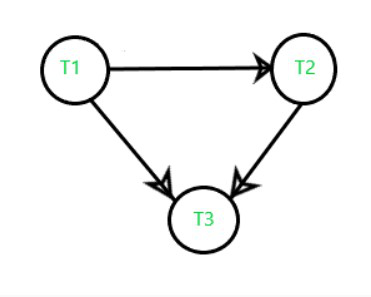


The above graph contains cycle/loop which means it is not conflict-serializable but it does not mean that it cannot be consistent and equivalent to the serial schedule it may or may not be.

**LookSchedule S’1 :**   
In the above example if we do **swapping** among some transaction’s operation so our table will look like –

| T1 | T2 | T3 |
| --- | --- | --- |
| a=100  **read(a) //100** |  |  |
| a=a-40  **write(a) //60** |  |  |
|  | a=a-40  **write(a) //20** |  |
|  |  | a=a-20  **write(a) //0** |

Its Precedence Graph is as follows – 



Now, we see that the precedence graph of the second table does not contain any cycle/loop, which means it is conflict serializable (equivalent to serial schedule, consistent) and the final result is coming the same as the first table.

## Conflict Serializable check

Lets check whether a schedule is conflict serializable or not. If a schedule is conflict Equivalent to its serial schedule then it is called Conflict Serializable schedule. Lets take few examples of schedules.

### Example of Conflict Serializability

Lets consider this schedule:

T1 T2

----- ------

R(A)

R(B)

R(A)

R(B)

W(B)

W(A)

To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Conflict Serializable**.

Lets take another example:

Lets take another example:

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

Lets **swap non-conflicting operations**:

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

T1 T2

----- ------

R(A)

R(A)

R(B)

W(B)

R(B)

W(A)

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

T1 T2

----- ------

R(A)

R(B)

R(A)

W(B)

R(B)

W(A)

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

T1 T2

----- ------

R(A)

R(B)

W(B)

R(A)

R(B)

W(A)

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