Chapter 4
Concurrence Control Techniques

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 database.

Types of locks

- **Binary locks**, which are simple, but are also too restrictive for database concurrency control purposes, and so are not used in practice.
- **Shared/exclusive locks—also** known as **read/write locks**—which provide more general locking capabilities and are used in practical database locking schemes.

Binary Locks

- A binary lock can have two states or values: **locked** and **unlocked** (or 1 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)
- A binary lock enforces mutual exclusion on the data item.

Lock and unlock operations for binary locks.

```
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;
```

- 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.

- 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.
  - A transaction T will not issue an unlock_item(X) operation unless it already holds the lock on item X.

- The DBMS has a lock manager subsystem to keep track of and control access to locks.
- The above 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

- Binary locks are too restrictive for database item
- We should allow several transactions to access the same item X if they all access X for reading purposes only.
- For this purpose **shared/exclusive** or **read/write** lock is used
  - There are three locking operations: **read_lock(X), write_lock(X), and unlock(X)**.
    - As before, each of the three locking operations should be considered indivisible
  - 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 transactions are 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.

Locking and unlocking operations for two mode (read-write or shared-exclusive) locks.

```
read_lock(X):
B:  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):
B:   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
                     end;
```

- When we use the shared/exclusive locking scheme, the system must enforce the following 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.

- A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T
- 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.
- 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.
- 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.

- Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own.
- Assume the transactions in the following figure (a) and their corresponding serial schedule result in figure (b)

(a)

T_1	T_2

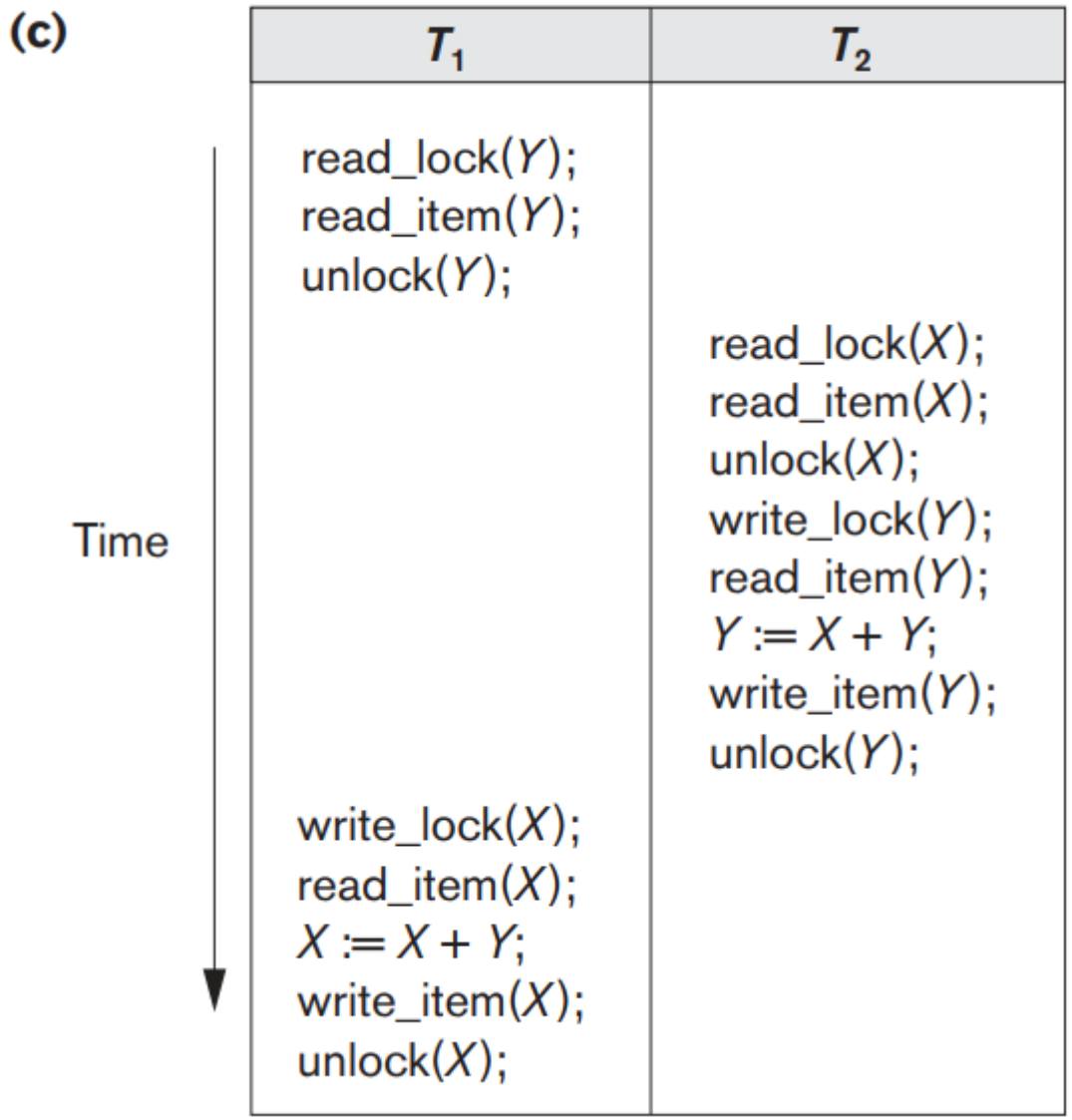
read_lock(Y);
read_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);
	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	$Y := X + Y$;
	write_item(Y);
	unlock(Y);

(b) Initial values: X=20, Y=30

Result serial schedule T_1 followed by T_2: X=50, Y=80

Result of serial schedule T_2 followed by T_1: X=70, Y=50

- The following is a possible schedule but non serializable.



Result of schedule *S*:
X=50, *Y*=50
(nonserializable)

Serializability by Two-Phase Locking

- A transaction is said to follow the **two-phase locking protocol (2PL)** if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- One of such 2PL is **basic 2PL** and the 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;
  - **a shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired.

Basic 2PL example

T_1'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'

read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

because T1' will issue its write_lock(X) before it unlocks item Y; consequently, when T2' issues its read_lock(X), it is forced to wait until T1' releases the lock by issuing an unlock (X) in the schedule.

These transactions follow the two-phase locking protocol. But they can produce a deadlock.

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.

Conservative 2PL (or static 2PL)

- **Conservative 2PL (or static 2PL)** requires a transaction to lock all the items it accesses before the transaction begins execution, by pre-declaring its read-set and write-set.
- If any of the pre-declared 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**

strict 2PL

- Strict 2PL, which guarantees strict schedules, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).
  - In strict 2PL 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.**

Rigorous 2PL

- Rigorous 2PL guarantees strict schedules.
- In rigorous 2PL 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.

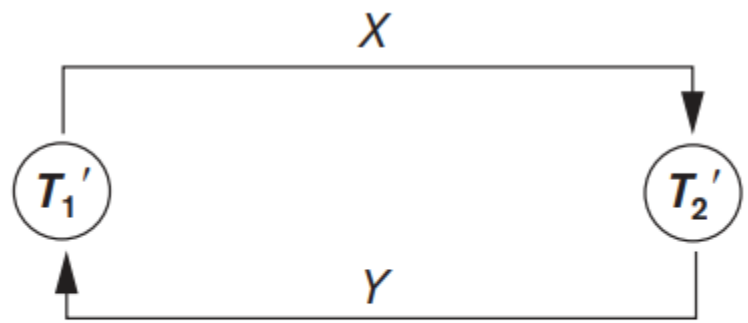
Dealing with Deadlock and Starvation

- Deadlock occurs when each transaction **T** in a set of two or more transactions is waiting 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 transactions in the set to release the lock on an item.

Time

T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

(b)



Deadlock Prevention Protocols

- **Conservative two-phase locking** helps in preventing deadlock even if it limits concurrency not a practical assumption.
- A **second protocol**, which also limits concurrency, involves ordering all the items in the database and making sure that a transaction that needs several items will lock them according to that order.
  - This requires that the programmer (or the system) is aware of the chosen order of the items, which is also not practical in the database context.

- Two schemes that prevent deadlock are called **wait-die and wound-wait.**
- Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock.
- The rules followed by these schemes are:
  - **Wait-die.** If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) Ti is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i dies) and restart it later with the same timestamp.
    - 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.

- **Wound-wait.** If $TS(T_i) < TS(T_j)$, then (T_i older than T_j) abort T_j (T_i wounds T_j) and restart it later with the same timestamp; otherwise (T_i younger than T_j) T_i is allowed to wait.
  - 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.
- These two techniques are **deadlock-free** but 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 immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.
- 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 T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock.
- The cautious waiting rules are as follows:
  - If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

Deadlock Detection

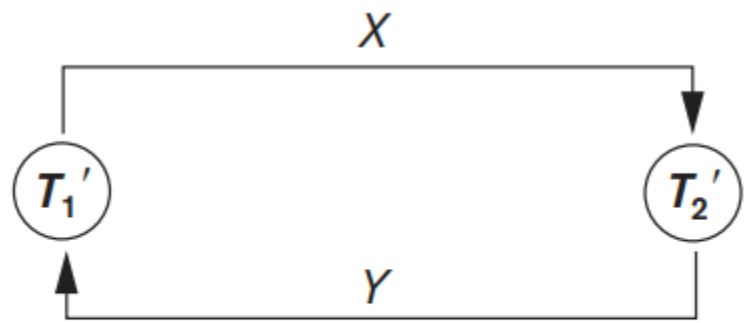
- Deadlock Detection more practical approach where the system checks if a state of deadlock actually exists.
- Suitable if the transactions are **short and each transaction locks only a few items**, or if the **transaction load is light.**
- If the transaction load is quite heavy, it may be advantageous 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 T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge ($T_i \rightarrow T_j$) is created in the wait-for graph.
  - When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph.
  - We have a state of deadlock if and only if the wait-for graph has a cycle

Time

T_1'	T_2'
read_lock(Y);	
read_item(Y);	
	read_lock(X);
	read_item(X);
write_lock(X);	
	write_lock(Y);

(b)



- 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

- 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.
- This method is practical because of its low overhead and simplicity.

Starvation

- Starvation is another problem that may occur when we use locking.
- 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
  - 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.

- 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 order in 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.

Concurrency Control Based on Timestamp Ordering

- 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.
- Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.
- Timestamps can be generated:
  - using a counter that is incremented each time its value is assigned to a transaction.
  - Using 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.

The Timestamp Ordering Algorithm

- 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 among all 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 timestamps 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 transaction T is aborted and resubmitted to the system as **a new transaction with a new timestamp.**
- It may result in **cascading rollback**

- The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering 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 operation. This should be done because some younger transaction with timestamp 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).

Multiversion Concurrency Control Techniques (**Reading assignment**)

- 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.
- It can be based on timestamp ordering and the other based on 2PL.

Multiversion Technique Based on Timestamp Ordering

- In this method, several versions X_1, X_2, ..., X_k of each data item X are maintained.
- For each version, the value of version X_i and the following two timestamps are kept:
  - read_TS(X_i). The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.
  - write_TS(X_i). The write timestamp of X_i is the timestamp of the transaction that wrote the value of version X_i.

- 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(X_i) of all versions of X that is also less than or equal to TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).
  - If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also less than or equal to TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

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.
- 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

A compatibility table for read/write/certify locking scheme.

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No