CONCURRENCY CONTROL TECHNIQUES

G C DIVYA

ASSISTANT PROFESSOR

CSE,JITD

Two-Phase Locking Techniques for Concurrency Control

- 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.
- There is one lock for each data item in the database.
- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Types of Locks and System Lock Tables

1. 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*)**.
- 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.

```
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;
```

- 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*.
- A binary lock enforces **mutual exclusion** on the data item.
- The lock_item and unlock_item operations must be implemented as indivisible units (known as **critical section** in operating systems).
- No interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits.
- 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 subsystem** to keep track of and control access to locks.

- **Binary locking scheme:**

1. A transaction *T* must issue the operation lock_item(*X*) before any read_item(*X*) or write_item(*X*) operations are performed in *T*.
2. A transaction *T* must issue the operation unlock_item(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T*.
3. A transaction *T* will not issue a lock_item(*X*) operation if it already holds the lock on item *X*.
4. A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

- Between the lock_item(*X*) and unlock_item(*X*) operations in transaction *T*, *T* is said to **hold the lock** on item *X*. At

2. 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.
- Each record in the lock table will have four fields: <Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>.
- 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 LOCK(*X*) = write-locked, the value of locking_transaction(s) is a *single transaction* that holds the exclusive (write) lock on *X*locked or write-locked, suitably coded.
- 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*.

```
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:

1. A transaction *T* must issue the operation read_lock(*X*) or write_lock(*X*) before any read_item(*X*) operation is performed in *T*.

2. A transaction *T* must issue the operation write_lock(*X*) before any write_item(*X*) operation is performed in *T*.

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

4. 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*.

5. 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*.

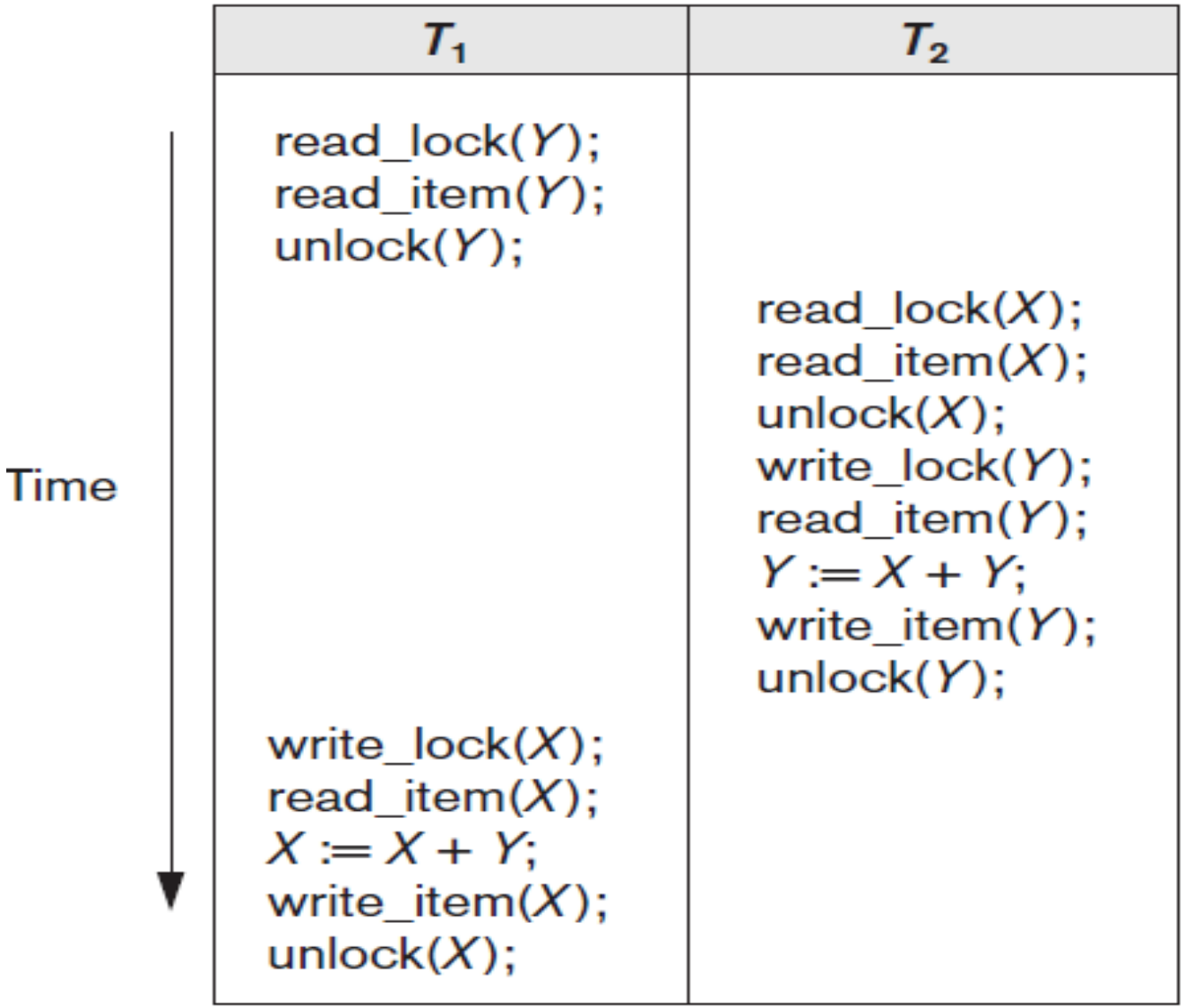
6. A transaction *T* will not issue an unlock(*X*) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

Conversion (Upgrading, Downgrading) of Locks

- 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.
- It is also possible for a transaction *T* to issue a write_lock(*X*) and then later to **downgrade** the lock by issuing a read_lock(*X*) operation.
- When upgrading and downgrading 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.

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);


T_1
T_2
read_lock(Y);
read_item(Y);
unlock(Y);
read_lock(X);
read_item(X);
unlock(X);
write_lock(Y);
read_item(Y);
Y := X + Y;
write_item(Y);
unlock(Y);
write_lock(X);
read_item(X);
X := X + Y;
write_item(X);
unlock(X);
Time

- **Drawbacks of Shared/Exclusive locks:**

1. May not sufficient to produce serializability schedule.
2. May not be free from non-recoverability.
3. May not be free from deadlock.
4. May not be free from starvation.

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.
- A transaction can be divided into two phases:

1. An **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released.
2. 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.

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);

- If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.
- Two-phase locking may limit the amount of concurrency.

Basic, Conservative, Strict, and Rigorous Two-Phase Locking (Cont.)

There are a number of variations of two-phase locking (2PL).

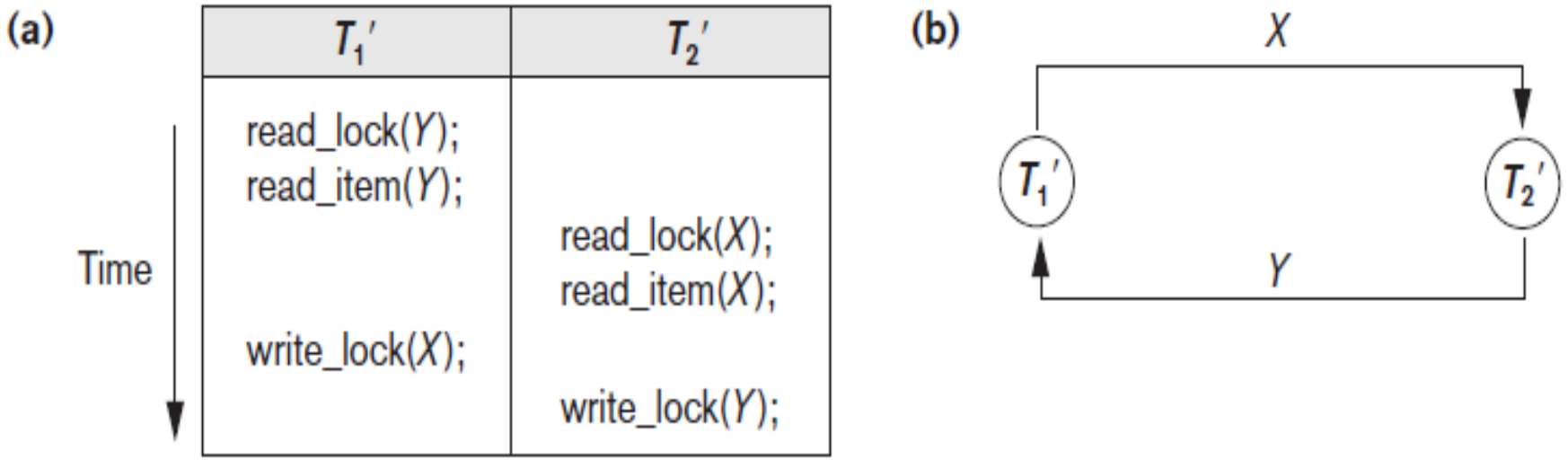
1. **Basic 2PL**
2. **Conservative 2PL:** Requires a transaction to lock all the items it accesses before the transaction begins execution, by **predeclaring** its read-set and write-set (Read_set and write_set) – deadlock free protocol.
3. **Strict 2PL:** A transaction T does not release any of its exclusive (write) locks until after it commits or aborts.
4. **Rigorous 2PL:** 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.

Differences: (Cont.)

1. **Strict and rigorous 2PL:** The former holds write-locks until it commits, whereas the latter holds all locks (read and write).
2. **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.

Dealing with Deadlock and Starvation (Cont.)

- **Deadlock** occurs when each transaction T1 in a set of two or more transactions is waiting for some item that is locked by some other transaction T2 in the set.
- 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. But because the other transaction is also waiting, it will never release the lock.



Deadlock Prevention Protocols (Cont.)

- **Deadlock Prevention Protocols:**
- One deadlock prevention protocol, which is used in conservative two-phase locking, requires that every transaction lock all the items it needs in advance, if any of the items cannot be obtained, none of the items are locked.
- The transaction waits and then tries again to lock all the items it needs.
- 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.
- 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?

- **Transaction timestamp** TS, which is a unique identifier assigned to each transact ion.
- The timestamps are typically based on the order in which transactions are started; hence, if transaction T1 starts before transaction T2, then TS(T1) < TS(T2).
- Notice that the older transaction (which starts first) has the smaller timestamp 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.** If $\mathrm{TS}(T_i) < \mathrm{TS}(T_j)$, then ($T_i$ older than T_j) T_i 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.*
  - **Wound-wait.** If $\mathrm{TS}(T_i) < \mathrm{TS}(T_j)$, 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.

- 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 immediately aborted 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.
- The cautious waiting rule is as follows:

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

Deadlock Detection:

- 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 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.

- 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.

- **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 in that it gives 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 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.
- 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.

Concurrency Control Based on Timestamp Ordering

1. Timestamps:

- A **timestamp** is a unique identifier created by the DBMS to identify a transaction.
- Timestamp of transaction T as **TS(T)**.
- Concurrency control techniques based on timestamp ordering do not use locks; hence, deadlocks cannot occur.
- The transaction timestamps are numbered 1, 2, 3, … in this scheme.
- A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time.

2. The Timestamp Ordering Algorithm for Concurrency Control:

- A schedule in which the transactions participate is then serializable, and the only **equivalent serial schedule** permitted has the transactions in order of their timestamp values. This is called **timestamp ordering (TO)**.
- The algorithm allows **interleaving** of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must follow the timestamp order.

T1	T2	T3
	R(A)	
	W(A)	
		R(C)
	W(B)	
		W(A)
		W(C)
R(A)		
R(B)		
W(A)		
W(B)		

Schedule S

- To do this, the algorithm associates with each database item *X* two timestamp (TS) values:

1. **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.
2. **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. Based on the algorithm, *T* will also be the last transaction to write item *X*.

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.
- If T is aborted and rolled back, any transaction T1 that may have used a value written by T must also be rolled back.
- Similarly, any transaction T2 that may have used a value written by T1 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.

- The concurrency control algorithm must check whether conflicting operations violate the timestamp ordering in the following two cases:

1. Whenever a transaction T issues a write_item(X) operation, the following check is performed:
   a. 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.
   b. 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).
2. Whenever a transaction T issues a read_item(X) operation, the following check is performed:
   a. 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.
   b. 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).

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 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 committed or aborted. This algorithm does not cause deadlock, since T waits for T′ only if TS(T) > TS(T′).

Thomas's Write Rule: It does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:

1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
2. 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).
3. 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

- These protocols for concurrency control keep copies of the old values of a data item when the item is updated (written); they are known as **Multiversion concurrency control** because several versions (values) of an item are kept by the system.
- One reason for keeping multiple versions 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, it writes a new version and the old version(s) of the item is retained.
- An obvious drawback of Multiversion techniques is that more storage is needed to maintain multiple versions of the database items.

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 associated with version Xi are kept:

1. **read_TS(Xi).** The **read timestamp** of Xi is the largest of all the timestamps of transactions that have successfully read version Xi.
2. **write_TS(Xi).** The **write timestamp** of Xi is the timestamp of the transaction that wrote the value of version Xi.

- To ensure serializability, the following rules are used:

1. 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).
2. 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 state of LOCK(*X*) for an item *X* can be one of read-locked, write-locked, certify-locked, or unlocked.
- Lock Compatibility table

(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

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

- 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, the **committed version**, must always have been written by some committed transaction. The second **local version** X′ can be created when a transaction T acquires a write lock on X.
- 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; this is another form of **lock upgrading**.
- 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.

Validation (Optimistic) Techniques and Snapshot Isolation Concurrency Control

- In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, no checking is done while the transaction is executing.
- **Validation-Based (Optimistic) Concurrency Control**:
- In this scheme, updates in the transaction are *not* applied directly to the database items on disk until the transaction reaches its end and is *validated*.
- At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializability.
- 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:

1. **Read phase.** A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
2. **Validation phase.** Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
3. **Write phase.** If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

- If there is little interference among transactions, most will be validated successfully.
- If there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later; under such circumstances, optimistic techniques do not work well.
- The techniques are called optimistic because they assume that little interference will occur and hence most transaction will be validated successfully, so that there is no need to do checking during transaction execution.

1. Transaction Tj completes its write phase before Ti starts its read phase.
2. Ti starts its write phase after Tj completes its write phase, and the read_set of Ti has no items in common with the write_set of Tj.
3. 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.

Concurrency Control Based on Snapshot Isolation:

- A transaction sees the data items that it reads based on the committed values of the items in the database snapshot (or database state) when the transaction starts.
- Snapshot isolation will ensure that the phantom record problem does not occur, since the database transaction, or, in some cases, the database statement, will only see the records that were committed in the database at the time the transaction started.
- Any insertions, deletions, or updates that occur after the transaction starts will not be seen by the transaction.
- In addition, snapshot isolation does not allow the problems of dirty read and nonrepeatable read to occur.

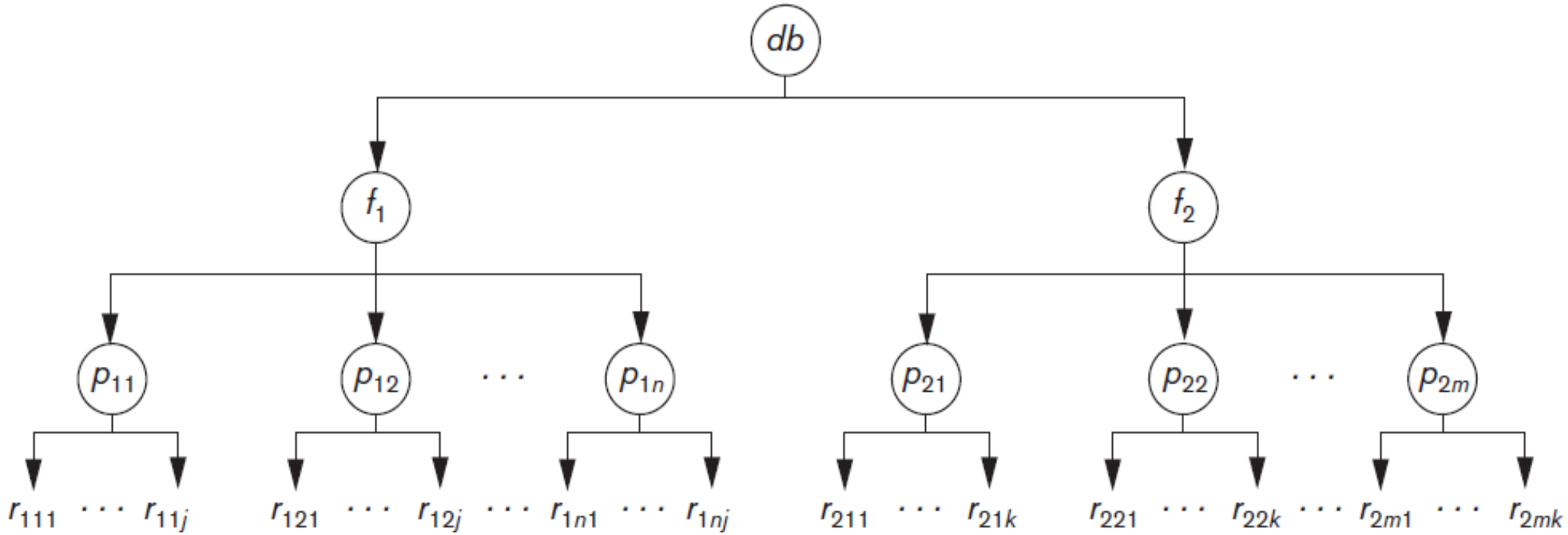
Granularity of Data Items and Multiple Granularity Locking

- All concurrency control techniques assume that the database is formed of a number of named data items.
- A database item could be chosen to be one of the following:
  - A database record
  - A field value of a database record
  - A disk block
  - A whole file
  - The whole database

Granularity Level Considerations for Locking

- The size of data items is often called the **data item granularity**.
- **Fine granularity** refers to small item sizes, whereas **coarse granularity** refers to large item sizes.
- Larger the data item size is, the lower the degree of concurrency permitted.
- The smaller the data item size is, the more the number of items in the database.

Multiple Granularity Level Locking



- To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed.

1. Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).
2. Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).
3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

The compatibility table of the three intention locks, and the actual shared and exclusive locks.

- The **multiple granularity locking (MGL)** protocol consists of the following rules:

1. The lock compatibility must be adhered to.
2. The root of the tree must be locked first, in any mode.
3. A node *N* can be locked by a transaction *T* in S or IS mode only if the parent node *N* is already locked by transaction *T* in either IS or IX mode.
4. A node *N* can be locked by a transaction *T* in X, IX, or SIX mode only if the parent of node *N* is already locked by transaction *T* in either IX or SIX mode.
5. A transaction *T* can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
6. A transaction *T* can unlock a node, *N*, only if none of the children of node *N* are currently locked by *T*.

T_1	T_2	T_3
IX(db)		
IX(f_1)		
	IX(db)	
		IS(db)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(p_{211})		
unlock(r_{211})		
unlock(p_{21})		
unlock(f_2)		
		S(f_2)
	unlock(p_{12})	
	unlock(f_1)	
	unlock(db)	
unlock(r_{111})		
unlock(p_{11})		
unlock(f_1)		
unlock(db)		
		unlock(r_{11j})
		unlock(p_{11})
		unlock(f_1)
		unlock(f_2)
		unlock(db)

- The multiple granularity level protocol is especially suited when processing a mix of transactions that include

1. short transactions that access only a few items (records or fields)
2. long transactions that access entire files.

- In this environment, less transaction blocking and less locking overhead are incurred by such a protocol when compared to a single-level granularity locking approach.

Using Locks for Concurrency Control in Indexes

- Two-phase locking can also be applied to B-tree and B+-tree indexes where the nodes of an index correspond to disk pages.
- Holding locks on index pages until the shrinking phase of 2PL could cause an undue amount of transaction blocking because searching an index always starts at the root.
- The tree structure of the index can be taken advantage of when developing a concurrency control scheme.
- A conservative approach for insertions would be to lock the root node in exclusive mode and then to access the appropriate child node of the root.
- If the child node is not full, then the lock on the root node can be released.
- An alternative, more **optimistic approach** would be to request and hold *shared* locks on the nodes leading to the leaf node, with an *exclusive* lock on the leaf.

- Another approach to index locking is to use a variant of the B+-tree, called the **B-link tree**.
- In a B-link tree, sibling nodes on the same level are linked at every level.
- For an insert operation, the shared lock on a node would be upgraded to exclusive mode.
- If a split occurs, the parent node must be relocked in exclusive mode.
- One complication is for search operations executed concurrently with the update.

Other Concurrency Control Issues

- **Insertion, Deletion, and Phantom Records:**
- When a new data item is **inserted** in the database, it obviously cannot be accessed until after the item is created and the insert operation is completed.
- In a locking environment, a lock for the item can be created and set to exclusive (write) mode; the lock can be released at the same time as other write locks would be released, based on the concurrency control protocol being used.
- For a timestamp-based protocol, the read and write timestamps of the new item are set to the timestamp of the creating transaction.
- For a **deletion operation** that is applied on an existing data item.
- For locking protocols, again an exclusive (write) lock must be obtained before the transaction can delete the item.
- For timestamp ordering, the protocol must ensure that no later transaction has read or written the item before allowing the item to be deleted

- A situation known as the **phantom problem** can occur when a new record that is being inserted by some transaction *T* satisfies a condition that a set of records accessed by another transaction *T′* must satisfy.
- The record that causes the conflict is a **phantom record** that has suddenly appeared in the database on being inserted.
- If other operations in the two transactions conflict, the conflict due to the phantom record may not be recognized by the concurrency control protocol.
- One solution to the phantom record problem is to use **index locking**.
- A more general technique, called **predicate locking**, would lock access to all records that satisfy an arbitrary *predicate* (condition) in a similar manner; however, predicate locks have proved to be difficult to implement efficiently

- **Interactive Transactions:**
- Another problem occurs when interactive transactions read input and write output to an interactive device, such as a monitor screen, before they are committed.
- The problem is that a user can input a value of a data item to a transaction *T* that is based on some value written to the screen by transaction *T′*, which may not have committed.
- This dependency between *T* and *T′* cannot be modeled by the system concurrency control method, since it is only based on the user interacting with the two transactions.

- **Latches**
- Locks held for a short duration are typically called **latches**. Latches do not follow the usual concurrency control protocol such as two-phase locking.
- For example, a latch can be used to guarantee the physical integrity of a disk page when that page is being written from the buffer to disk.
- A latch would be acquired for the page, the page written to disk, and then the latch released.