

# Purpose of Concurrency Control


- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Example: In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

# Two-Phase Locking Techniques

Locking is an operation which secures (a) permission to Read or (b) permission to Write a data item for a transaction. Example: Lock (X). Data item X is locked in behalf of the requesting transaction.

Unlocking is an operation which removes these permissions from the data item. Example: Unlock (X). Data item X is made available to all other transactions. Lock and Unlock are Atomic operations.



# Types of Locks

(I) **Binary Lock:** have two states or values: locked and unlocked (or 1 and 0).

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. We refer to the current value (or state) of the lock associated with item  $X$  as  $\text{lock}(X)$ .

Two operations,  $\text{lock\_item}$  and  $\text{unlock\_item}$ , are used with binary locking and these two operations must be implemented as indivisible units; that is, no interleaving should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits.

# Binary Lock....

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

These rules must 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 transaction can access the same item concurrently.

# Binary Lock ...

Figure: Lock and unlock operations for binary locks.

➤ The following code performs the lock operation:

```
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);  
    goto B  
end;
```

➤ The following code performs the unlock operation:

```
LOCK (X) ← 0 (*unlock the item*)  
if any transactions are waiting then  
    wake up one of the waiting transactions;
```

## (II) Shared/Exclusive (or Read/Write) Locks

There are three locking operation: `read_lock(x)`, `write_lock(x)`, and `unlock(x)`.

Two lock modes (a) shared (read) and (b) exclusive (write).

Shared mode: shared lock (X). More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.

Exclusive mode: Write lock (X). Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

# Shared/Exclusive Lock..

- When we use 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 a 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.

# Shared/Exclusive Lock..

The following code performs the read operation:

```
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;
```



# Shared/Exclusive Lock..

The following code performs the write lock operation:

```
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;
```

# Shared/Exclusive Lock..

The following code performs the unlock operation:

```
if LOCK (X) = "write-locked" then
    begin LOCK (X) ← "unlocked";
        wakes up one of the transactions, if any
    end
else if 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";
                wake up one of the transactions, if any
            end
        end
    end;
end;
```

# Shared/Exclusive Lock..

## ➤ Lock conversion

Sometime it is desirable to relax conditions 4 and 5 in the shared/exclusive lock rule in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain condition to convert the lock from one locked state to another.

**Lock upgrade:** from existing read lock to write lock.

if  $T_i$  has a read-lock (X) and  $T_j$  has no read-lock (X) ( $i \neq j$ ) then  
convert read-lock (X) to write-lock (X)

else

force  $T_i$  to wait until  $T_j$  unlocks X

**Lock downgrade:** from existing write lock to read lock.

if  $T_i$  has a write-lock (X) (\*no transaction can have any lock on X\*)  
convert write-lock (X) to read-lock (X)

# Two-Phase Locking Techniques: Essential components

Lock Manager: Managing locks on data items.

Lock table: Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Database requires that all transactions should be well-formed.

A transaction is well-formed if:

- It must lock the data item before it reads or writes to it.
- It must not lock an already locked data items and it must not try to unlock a free data item.

# Two-Phase Locking Techniques: The algorithm

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.

**Two Phases:** (a) Locking (Growing) (b) Unlocking (Shrinking).

**Locking (Growing) Phase:** A transaction applies locks (read or write) on desired data items one at a time.

**Unlocking (Shrinking) Phase:** A transaction unlocks its locked data items one at a time.

**Requirement:** For a transaction these two phases must be mutually exclusive, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

If every transaction in a schedule follows the 2PL, the schedule is guaranteed to be serializable, obviating the need to test for serializability of schedule any more. The locking mechanism, by enforcing two-phase locking rules, also enforce serializability.

# Transactions not obeying 2PL

## T1

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

## T2

```
read_lock (X);  
read_item (X);  
unlock (X);  
Write_lock (Y);  
read_item (Y);  
Y:=X+Y;  
write_item (Y);  
unlock (Y);
```


## Result

Initial values: X=20; Y=30  
Result of serial execution  
T1 followed by T2  
X=50, Y=80.  
Result of serial execution  
T2 followed by T1  
X=70, Y=50

# Transactions not obeying 2PL

T1	T2	<u>Result</u>
read_lock (Y); read_item (Y); <b>unlock (Y);</b>	read_lock (X); read_item (X); <b>unlock (X);</b> <b>write_lock (Y);</b> read_item (Y); Y:=X+Y; write_item (Y); unlock (Y);	X=50; Y=50 Nonserializable because it violates two-phase policy.
<b>write_lock (X);</b> read_item (X); X:=X+Y; write_item (X); unlock (X);		

Time



# Two-Phase Locking Techniques: The algorithm

## T1'

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

## T2'

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

T1 and T2 follow two-phase policy. But they are subject to deadlock, which must be dealt with.





# Two-Phase Locking Techniques: The algorithm

Two-phase policy generates two locking algorithms

(a) Basic and (b) Conservative.

Conservative: Prevents deadlock by locking all desired data items before transaction begins execution. If all desired items cannot be locked, the transaction waits.

Basic: Transaction locks data items incrementally. This may cause deadlock which is dealt with.

Strict: A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.



# 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 on a waiting queue, waiting for one of the other transactions in the set to release the lock on an item.

T1'

read\_lock (Y);  
read\_item (Y);

write\_lock (X);  
(waits for X)

T2'

read\_lock (X);  
read\_item (Y);

write\_lock (Y);  
(waits for Y)

T1 and T2 did follow two-phase policy but they are deadlock

**Deadlock (T1' and T2')**

# Dealing with Deadlock and Starvation

## Deadlock prevention

A transaction locks all data items it refers to before it begins execution. This way of locking prevents deadlock since a transaction never waits for a data item. The **conservative two-phase locking** uses this approach. This solution obviously 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 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? These techniques use the concept of **transaction timestamp**  $TS(T)$ , which is a unique identifier assigned to each transaction depending on when the transaction started. Two schemes that prevent deadlock are called

**wait-die** and **wound-wait**. Rules followed by these schemes are as follows:

**Wait-die:** if  $TS(T_i) < TS(T_j)$ , 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.

## Deadlock prevention ...

**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.

In wait-die, an older transaction is allowed to wait on 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 on 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 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 protocol that prevent deadlock do not require timestamps are no waiting (NW) and cautious waiting (CW) algorithms. In the **no waiting (NW)**

**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.

**Cautious waiting (CW)** : Because **NW** scheme can cause transactions to abort and restart needlessly, the **CW** algorithm was proposed to try to reduce the number of needless abort/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 CW 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  $T_i$ .

# Dealing with Deadlock and Starvation

## Deadlock detection and resolution

In this approach, deadlocks are allowed to happen. The scheduler maintains a **wait-for-graph** for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.

A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like:  $T_i$  waits for  $T_j$  waits for  $T_k$  waits for  $T_i$  or  $T_j$  occurs, then this creates a cycle. One of the transaction of the cycle is selected and rolled back and is called **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.


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.

# Dealing with Deadlock and Starvation

## Starvation

Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further. In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back. This limitation is inherent in all priority based scheduling mechanisms. In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

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.





# Timestamp based concurrency control algorithm

## Timestamp

A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation. Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

Each database item  $X$  has two timestamp (TS) valuse:

1.  $\text{read\_TS}(X)$ : The **read timestamp** of item  $X$ ; this is the largest timestamp among all the timestamps of transactions that have successfully read item  $X$  – that is,  $\text{read\_TS}(X) = \text{TS}(T)$ , where  $T$  is the youngest transaction that has read  $X$  successfully.
2.  $\text{write\_TS}(X)$ : The **write timestamp** of item  $X$ ; this is largest of all the timestamps of transactions that have successfully written item  $X$  – that is,  $\text{write\_TS}(X) = \text{TS}(T)$ , where  $T$  is the youngest transaction that has written  $X$  successfully.

# Timestamp based concurrency control algorithm

## Basic Timestamp Ordering

### 1. Transaction T issues a **write\_item(X)** operation:

- a. If  $\text{read\_TS}(X) > \text{TS}(T)$  or if  $\text{write\_TS}(X) > \text{TS}(T)$ , then a younger transaction has already read or write the data item X before T had a chance to write X, so abort and roll-back T and reject the operation.
- b. If the condition in part (a) does not exist, then execute the **write\_item(X)** of T and set  $\text{write\_TS}(X)$  to  $\text{TS}(T)$

### 2. Transaction T issues a **read\_item(X)** operation:

- a. If  $\text{write\_TS}(X) > \text{TS}(T)$ , then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
- b. If  $\text{write\_TS}(X) \leq \text{TS}(T)$ , then execute **read\_item(X)** of T and set  $\text{read\_TS}(X)$  to the larger of  $\text{TS}(T)$  and the current  $\text{read\_TS}(X)$ .

The Basic TO guaranteed to be conflict serializable, like the 2PL protocol. In TO, the deadlock does not occur. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.



# Timestamp based concurrency control algorithm

## Strict Timestamp Ordering

### 1. Transaction T issues a `write_item(X)` operation:

If  $TS(T) > read\_TS(X)$ , then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

### 2. Transaction T issues a `read_item(X)` operation:

If  $TS(T) > write\_TS(X)$ , then delay T until the transaction T' that wrote or read X has terminated (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 is a modification of the basic TO algorithm and 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 just ignore (do not execute) the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
3. If the conditions given in 1 and 2 above do not occur, then execute `write_item(X)` of T and set  $write\_TS(X)$  to  $TS(T)$ .

# Multiversion concurrency control techniques

## Concept

This approach maintains a number of versions of a data item and allocates the right version to a read operation of a transaction. Thus unlike other mechanisms a read operation in this mechanism is never rejected.

Side effect: Significantly more storage (RAM and disk) is required to maintain multiple versions. To check unlimited growth of versions, a garbage collection is run when some criteria is satisfied.



# Multiversion technique based on timestamp ordering

Assume  $X_1, X_2, \dots, X_n$  are the version of a data item  $X$  created by a write operation of transactions. With each  $X_i$  a  $\text{read\_TS}$  (read timestamp) and a  $\text{write\_TS}$  (write timestamp) are associated.

**$\text{read\_TS}(X_i)$ :** The read timestamp of  $X_i$  is the largest of all the timestamps of transactions that have successfully read version  $X_i$ .

**$\text{write\_TS}(X_i)$ :** The write timestamp of  $X_i$  that wrote the value of version  $X_i$ .

A new version of  $X_i$  is created only by a write operation.

**To ensure serializability, the following two rules are used.**

1. If transaction  $T$  issues  $\text{write\_item}(X)$  and version  $i$  of  $X$  has the highest  $\text{write\_TS}(X_i)$  of all versions of  $X$  that is also less than or equal to  $\text{TS}(T)$ , and  $\text{read\_TS}(X_i) > \text{TS}(T)$ , then abort and roll-back  $T$ ; otherwise create a new version  $X_j$  and  $\text{read\_TS}(X_j) = \text{write\_TS}(X_j) = \text{TS}(T)$ .
2. If transaction  $T$  issues  $\text{read\_item}(X)$ , find the version  $i$  of  $X$  that has the highest  $\text{write\_TS}(X_i)$  of all versions of  $X$  that is also less than or equal to  $\text{TS}(T)$ , then return the value of  $X_i$  to  $T$ , and set the value of  $\text{read\_TS}(X_i)$  to the largest of  $\text{TS}(T)$  and the current  $\text{read\_TS}(X_i)$ .

Rule 2 guarantees that a read will never be rejected.

# Granularity of data items and Multiple Granularity Locking

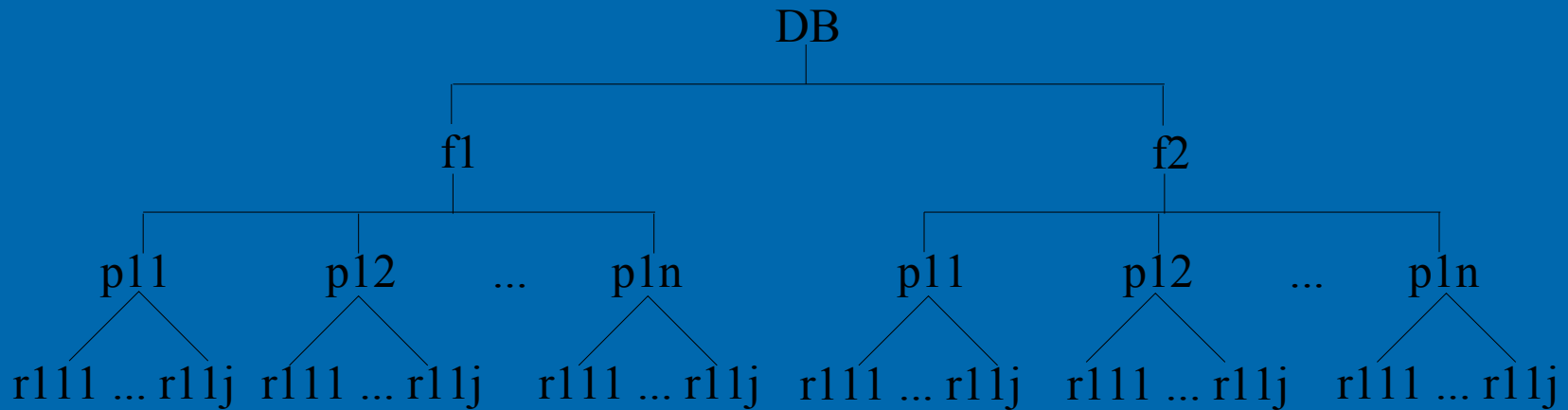
A lockable unit of data defines its granularity. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation). Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and high for fine granularity. Example of data item granularity:

1. A field of a database record (an attribute of a tuple).
2. A database record (a tuple or a relation).
3. A disk block.
4. An entire file.
5. The entire database.



# Granularity of data items and Multiple Granularity Locking

The following diagram illustrates a hierarchy of granularity from coarse (database) to fine (record).



# Granularity of data items and Multiple Granularity Locking

To manage such hierarchy, in addition to read and write, three additional locking modes, called intention lock modes are defined:

**Intention-shared (IS):** indicates that a shared lock(s) will be requested on some descendent nodes(s).

**Intention-exclusive (IX):** indicates that an exclusive lock(s) will be requested on some descendent nodes(s).

**Shared-intention-exclusive (SIX):** indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendent nodes(s).

These locks are applied using the following compatibility matrix:

	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

# Granularity of data items and Multiple Granularity Locking

The set of rules which must be followed for producing serializable schedule are:

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 is already locked by T in either IS or IX mode.
4. A node N can be locked by T in X, IX, or SIX mode only if the parent of N is already locked by T in either IX or SIX mode.
5. T can lock a node only if it has not unlocked any node (to enforce 2PL policy).
6. T can unlock a node, N, only if none of the children of N are currently locked by T.

# Granularity of data items and Multiple Granularity Locking

An example of a serializable execution:

T1	T2	T3
IX(db)		
IX(f1)		
	IX(db)	
		IS(db)
		IS(f1)
		IS(p11)
IX(p11)		
X(r111)		
	IX(f1)	
	X(p12)	
		S(r11j)
IX(f2)		
IX(p21)		
IX(r211)		
Unlock (r211)		
Unlock (p21)		
Unlock (f2)		
		S(f2)