

# Chapter 5

## **Concurrency Control Techniques**

Adapted from the slides of “Fundamentals of Database Systems” (Elmasri et al., 2006)

---

# Chapter Outline

- Purpose of Concurrency Control
- Two-Phase Locking Techniques
- Concurrency Control Based on Timestamp Ordering
- Multi-version Concurrency Control Techniques
- Validation (Optimistic) Concurrency Control Techniques
- Granularity of Data Items And Multiple Granularity Locking

# 1. 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 concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

## 2. Two-Phase Locking Techniques (1)

- Locking is an operation which secures
  - (a) permission to Read
  - (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.

# Two-Phase Locking Techniques (2)

- 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 (3)

- Type of Locks:
  - Binary Locks
  - Shared/ Exclusive (or Read/ Write) Locks

# Two-Phase Locking Techniques (4)

## ■ Binary Locks

- 2 values: locked and unlocked (1 and 0)
- 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;
```

# Two-Phase Locking Techniques (5)

- Binary Locks

- The following code performs the unlock operation:

LOCK (X)  $\leftarrow$  0 (\*unlock the item\*)

if any transactions are waiting then

wake up one of the waiting the transactions;



# Two-Phase Locking Techniques (6)

## ■ Binary Locks

### □ Rules:

1. A transaction  $T$  must issue the operation `lock_item(X)` before any `read_item(X)` or `write_item(X)` operations 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  $X$ .

# Two-Phase Locking Techniques (7)

- Shared/ Exclusive (or Read/ Write) Locks
  - Two locks modes:
    - (a) shared (read) (b) exclusive (write).
  - **Shared mode:** read 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.

# Two-Phase Locking Techniques (8)

- Shared/ Exclusive (or Read/ Write) Locks
  - Lock Manager:
    - Managing locks on data items.
  - Lock table:
    - Lock manager uses it to store the identify 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

# Two-Phase Locking Techniques (9)

- Shared/ Exclusive (or Read/ Write) Locks

- The following code performs the **read lock** 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;
```

# Two-Phase Locking Techniques (10)

- Shared/ Exclusive (or Read/ Write) Locks
  - 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;
```

# Two-Phase Locking Techniques (11)

- Shared/ Exclusive (or Read/ Write) Locks
  - 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;
```

# Two-Phase Locking Techniques (12)

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

### □ 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  $T$ .

# Two-Phase Locking Techniques (13)

- Shared/ Exclusive (or Read/ Write) Locks

- Rules (cont.):

- 4. A transaction  $T$  must 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  $T$  must 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  $T$  must not issue the operation *unlock(X)* unless it already holds a read (shared) lock or a write(exclusive) lock on item  $X$ .



# Two-Phase Locking Techniques (14)

- Shared/ Exclusive (or Read/ Write) Locks

- Lock conversion

- **Lock upgrade:** 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:** existing write lock to read lock

$T_i$  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 (15)

## ■ Two-Phase Locking

- 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 exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

# Two-Phase Locking Techniques (16)

## ■ Two-Phase Locking

$T_1$	$T_2$
<pre>read_lock(Y); read_item(Y); unlock(Y); write_lock(X); read_item(X); X := X + Y; write_item(X); unlock(X);</pre>	<pre>read_lock(X); read_item(X); unlock(X); write_lock(Y); read_item(Y); Y := X + Y; write_item(Y); unlock(Y);</pre>

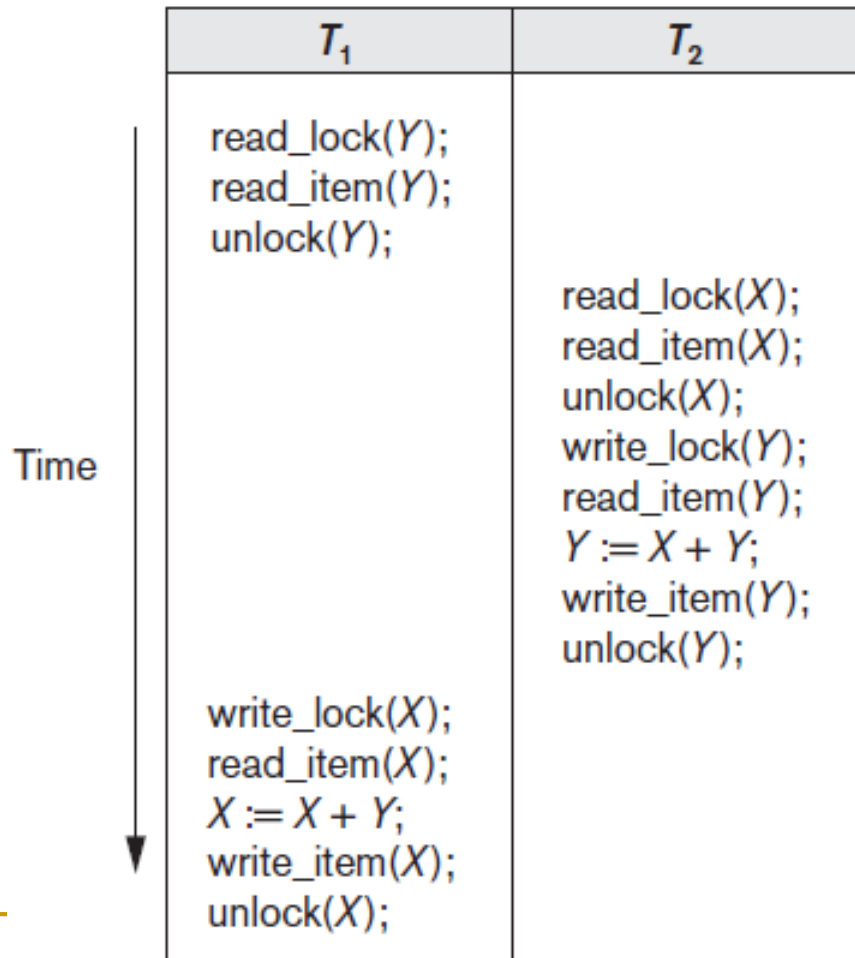
Initial values:  $X=20$ ,  $Y=30$

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

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

# Two-Phase Locking Techniques (17)

## ■ Two-Phase Locking



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

# Two-Phase Locking Techniques (18)

## ■ Two-Phase Locking

$T_1'$	$T_2'$
<pre>read_lock(Y); read_item(Y); write_lock(X); unlock(Y) read_item(X); X := X + Y; write_item(X); unlock(X);</pre>	<pre>read_lock(X); read_item(X); write_lock(Y); unlock(X) read_item(Y); Y := X + Y; write_item(Y); unlock(Y);</pre>

$T_1'$  and  $T_2'$  follow two-phase policy but they are subject to deadlock, which must be dealt with.

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);

wait

read\_lock (X);  
read\_item (X);  
**write\_lock (Y);**  
unlock (X);  
read\_item (Y);  
Y:=X+Y;  
write\_item (Y);  
unlock (Y);

**Guaranteed to be  
serializable**

T'1

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

Deadlock

**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);~~

**Can produce a deadlock**

# Two-Phase Locking Techniques (19)

- Two-Phase Locking

- Variations:

- (a) **Basic**
    - (b) **Conservative**
    - (c) **Strict**
    - (d) **Rigorous**

- **Conservative:**

- Prevents deadlock by locking all desired data items before transaction begins execution.

- **Basic:**

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



# Two-Phase Locking Techniques (20)

## ■ Two-Phase Locking

### □ **Strict:**

- A transaction T does not release any of its *exclusive (write) locks* until *after* it commits or aborts.
- The most commonly used two-phase locking algorithm.

### □ **Rigorous:**

- A Transaction T does not release any of its *locks (Exclusive or shared)* until after it commits or aborts.

# Two-Phase Locking Techniques (21)

## ■ Dealing with Deadlock and Starvation

### □ **Deadlock**

**$T'_1$**

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

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

**$T'_2$**

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

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

$T'_1$  and  $T'_2$  did follow two-phase policy but they are deadlock

### □ Deadlock ( $T'_1$ and $T'_2$ )

# Two-Phase Locking Techniques (22)

## ■ 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.

# Two-Phase Locking Techniques (23)

- 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.

# Two-Phase Locking Techniques (24)

## ■ Dealing with Deadlock and Starvation

### □ **Deadlock detection and resolution**

#### ■ A wait-for-graph:

- One node is 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.
- When  $T_j$  releases the lock(s) on the items that  $T_i$  was waiting for, the directed edge is dropped.
- We have a state of deadlock if and only if the wait-for graph has a cycle.

- When the system should check for a deadlock?

# Two-Phase Locking Techniques (25)

$\underline{T'_1}$

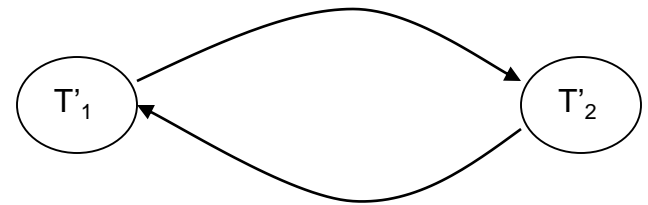
read\_lock (Y);  
read\_item (Y);

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

$\underline{T'_2}$

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

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



b) wait-for graph

a) Partial schedule of  $T'_1$  and  $T'_2$

# Two-Phase Locking Techniques (26)

## ■ Dealing with Deadlock and Starvation

### □ **Deadlock avoidance**

- There are many variations of two-phase locking algorithm.
- Some avoid deadlock by not letting the cycle to complete.
- That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
- Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

# Two-Phase Locking Techniques (27)

## ■ Dealing with Deadlock and Starvation

### □ **Deadlock avoidance**

#### ■ Timestamp:

- $TS(T)$
- A unique identifier assigned to each transaction.
- Typically based on the order in which transactions are started
- 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* timestamp value.



# Two-Phase Locking Techniques (28)

## ■ Dealing with Deadlock and Starvation

### □ Deadlock avoidance

#### ■ 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***.

#### ■ 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.

# Two-Phase Locking Techniques (29)

## ■ 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.
- ❑ Wound-Wait and wait-die scheme can avoid starvation.

### 3. Concurrency Control Based on Timestamp Ordering (1)

#### ■ **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.

# Concurrency Control Based on Timestamp Ordering (2)

## ■ Timestamp

- The algorithm associates with each database item  $X$  with two timestamp (TS) values:
  - $Read\_TS(X)$ : The **read timestamp** of item  $X$ ; this is the largest timestamp among all the timestamps of transactions that have successfully read item  $X$ .
  - $Write\_TS(X)$ : The **write timestamp** of item  $X$ ; this is the largest timestamp among all the timestamps of transactions that have successfully written item  $X$ .

# Concurrency Control Based on Timestamp Ordering (3)

## ■ Basic Timestamp Ordering

- 1. Transaction T issues a write\_item(X) operation:
  - (a) If  $\text{read\_TS}(X) > \text{TS}(T)$  or if  $\text{write\_TS}(X) > \text{TS}(T)$ 
    - an younger transaction has already read the data item
    - abort and roll-back T **with a new timestamp** and reject the operation.
  - (b) If the condition in part (a) does not exist, then execute write\_item(X) of T and set write\_TS(X) to TS(T).
- 2. Transaction T issues a read\_item(X) operation:
  - (a) If  $\text{write\_TS}(X) > \text{TS}(T)$ 
    - an younger transaction has already written to the data item
    - abort and roll-back T **with a new timestamp** and reject the operation.
  - (b) If  $\text{write\_TS}(X) \leq \text{TS}(T)$ , then execute read\_item(X) of T and set read\_TS(X) to the larger of (TS(T) and the current read\_TS(X) )

## Example: Three transactions executing under a timestamp-based scheduler

T1	T2	T3	A	B	C
200	150	175	RT = 0 WT = 0	RT = 0 WT = 0	RT = 0 WT = 0
r1(B)  w1(B) w1(A)	r2(A)   w2(C) <b>Abort</b>	r3(C)   w3(A)	RT = 150   WT = 200	RT = 200   WT = 200	RT = 175

Why T2 must be aborted (rolled-back)?

# Concurrency Control Based on Timestamp Ordering (4)

## ■ Strict Timestamp Ordering

- 1. Transaction T issues a write\_item(X) operation:
  - If  $TS(T) > write\_TS(X)$ , then delay T until the transaction T' that wrote 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 X has terminated (committed or aborted).
- Ensures the schedules are both strict and conflict serializable

# Concurrency Control Based on Timestamp Ordering (5)

## ■ Thomas's Write Rule

Modify the checks for the write\_item(X) operation:

- ❑ 1. If  $\text{read\_TS}(X) > \text{TS}(T)$  then abort and roll-back T and reject the operation.
- ❑ 2. If  $\text{write\_TS}(X) > \text{TS}(T)$ , then just **ignore** the write operation and continue execution because it is already outdated and obsolete.
- ❑ 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).



## 4. Multiversion Concurrency Control Techniques (1)

- 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 Concurrency Control Techniques (2)

- **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 `read_TS` (read timestamp) and a `write_TS` (write timestamp) are associated.
  - **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  $X_i$ .
  - A new version of  $X_i$  is created only by a write operation.

# Multiversion Concurrency Control Techniques (3)

## ■ Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used:

1. If transaction  $T$  issues `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  $X_i$  and  $\text{read\_TS}(X) = \text{write\_TS}(X_i) = \text{TS}(T)$ .
2. If transaction  $T$  issues `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.

## Example: Execution of transactions using multiversion concurrency control

T1	T2	T3	T4	A <sub>0</sub>	A <sub>150</sub>	A <sub>200</sub>
150	200	175	225			
r1(A) w1(A)	r2(A) w2(A)	r3(A)	r4(A)	read	Create Read  read	Create  read

**Note:** T3 does not have to abort, because it can read an earlier version of A.

# Multiversion Concurrency Control Techniques (4)

## Multiversion Two-Phase Locking Using Certify Locks

### ■ Concept:

- Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T.
- This is accomplished by **maintaining two versions of each data item X**
  - One version must always have been written by some committed transaction. This means a write operation always creates a new version of X.
  - The second version created when a transaction acquires a write lock on the item.

# Multiversion Concurrency Control Techniques (5)

## Multiversion Two-Phase Locking Using Certify Locks

### ■ Steps:

1. X is the committed version of a data item.
2. T creates a second version X' after obtaining a write lock on X.
3. Other transactions continue to read X.
4. T is ready to commit so it obtains a certify lock on X'.
5. The committed version X becomes X'.
6. T releases its certify lock on X', which is X now.

Compatibility tables for

	Read	Write
Read	yes	no
Write	no	no

read/write locking scheme

	Read	Write	Certify
Read	yes	no	no
Write	no	no	no
Certify	no	no	no

read/write/certify locking scheme

# Multiversion Concurrency Control Techniques (6)

## Multiversion Two-Phase Locking Using Certify Locks

### ■ Note:

- ❑ In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently.
- ❑ This **improves concurrency** but it **may delay transaction commit** because of obtaining certify locks on all its writes. It avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked.

# 5. Validation (Optimistic)

## Concurrency Control Techniques (1)

- In this technique only at the time of commit serializability is checked and transactions are aborted in case of non-serializable schedules.
- Three phases:
  1. Read phase
  2. Validation phase
  3. Write phase

### 1. Read phase:

- A transaction can read values of committed data items. However, updates are applied only to **local copies** (versions) of the data items (in database cache).



# Validation (Optimistic) Concurrency Control Techniques (2)

2. **Validation phase:** Serializability is checked before transactions write their updates to the database.

- This phase for  $T_i$  checks that, for each transaction  $T_j$  that is either committed or is in its validation phase, one of the following conditions holds:
  1.  $T_j$  completes its write phase before  $T_i$  starts its read phase.
  2.  $T_i$  starts its write phase after  $T_j$  completes its write phase, and the read\_set of  $T_i$  has no items in common with the write\_set of  $T_j$
  3. Both the read\_set and write\_set of  $T_i$  have no items in common with the write\_set of  $T_j$ , and  $T_j$  completes its read phase before  $T_i$  completes its read phase.
- The first condition is checked first for each transaction  $T_j$ . If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds,  $\rightarrow$  fails and  $T_i$  is aborted.

---

# Validation (Optimistic) Concurrency Control Techniques (3)

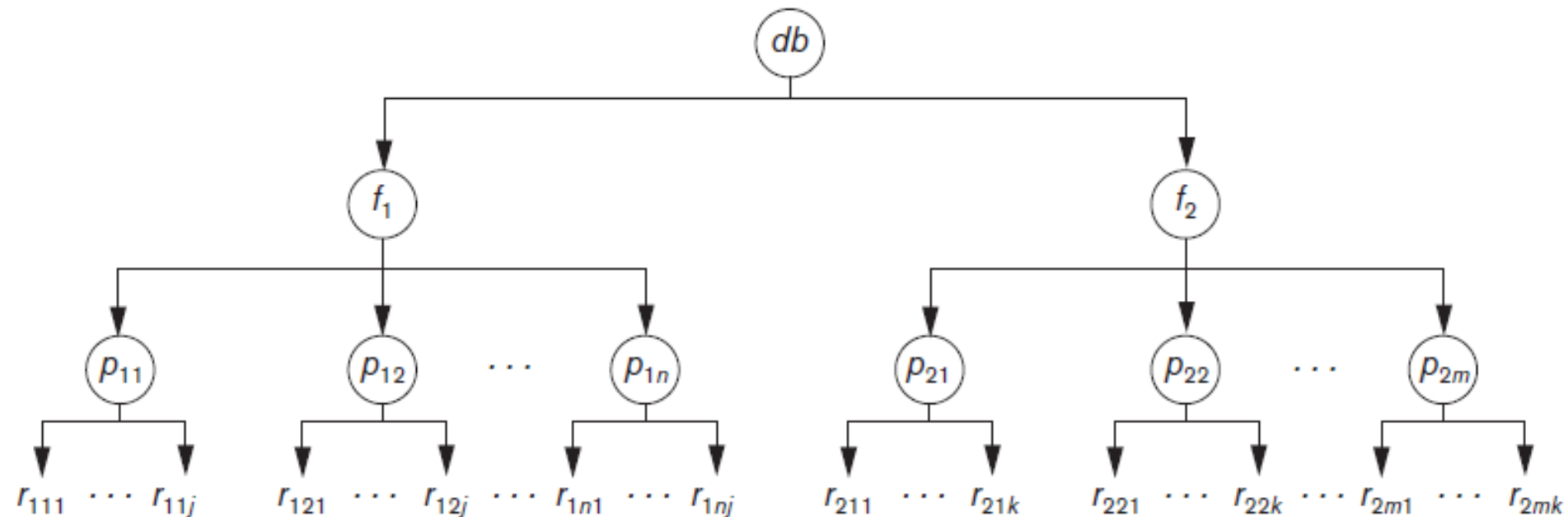
3. **Write phase:** On a successful validation transactions' updates are applied to the database; otherwise, transactions are restarted.

## 6. Granularity of Data Items And Multiple Granularity Locking (1)

- 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 (2)

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



# Granularity of data items and Multiple Granularity Locking (3)

- 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 node(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).

# Granularity of data items and Multiple Granularity Locking (4)

- These locks are applied using the following compatibility matrix:

	IS	IX	S	SIX	X	
IS	Yes	Yes	Yes	Yes	No	Intention-shared (IS) Intention-exclusive (IX) Shared-intention-exclusive (SIX)
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 (5)

- The set of rules which must be followed for producing serializable schedule:
  1. The lock compatibility must 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 IX 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 (6)

- An example of a serializable execution:

$T_1$	$T_2$	$T_3$
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( $p_{211}$ )		

$T_1$  wants to update  $r_{111}$ ,  $r_{211}$   
 $T_2$  wants to update all records on page  $p_{12}$   
 $T_3$  wants to read  $r_{11j}$  and the entire file  $f_2$



# Granularity of data items and Multiple Granularity Locking (7)

- An example of a serializable execution (continued):

$T_1$	$T_2$	$T_3$
$\text{unlock}(r_{211})$ $\text{unlock}(p_{21})$ $\text{unlock}(f_2)$	$\text{unlock}(p_{12})$ $\text{unlock}(f_1)$ $\text{unlock}(db)$	$S(f_2)$
$\text{unlock}(r_{111})$ $\text{unlock}(p_{11})$ $\text{unlock}(f_1)$ $\text{unlock}(db)$		$\text{unlock}(r_{11j})$ $\text{unlock}(p_{11})$ $\text{unlock}(f_1)$ $\text{unlock}(f_2)$ $\text{unlock}(db)$