Chapter 5

Concurrency Control Techniques

Adapted from the slides of “Fundamentals of Database Systems” (Elmasri et al., 2003)

Outline

Databases Concurrency Control

Database Concurrency Control

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 T_1 conflicts with T_2 over a data item A, then the existing concurrency control decides if T_1 or T_2 should get the A and if the other transaction is rolled-back or waits.

Database Concurrency Control

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.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

Two locks 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

Database Concurrency Control

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 (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 Concurrency Control

Two-Phase Locking Techniques: Essential components

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.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

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;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

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;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

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;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

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;
```

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

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;
```

When we use the share/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* 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*.

Database Concurrency Control

Two-Phase Locking Techniques: Essential components

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)
```

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

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.

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

<u>T1</u>

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

<u>T2</u>

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

<u>Result</u>

Initial values: X=20; Y=30
Result of serial execution
T_1 followed by T_2
X=50, Y=80.
Result of serial execution
T_2 followed by T_1
X=70, Y=50

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Time

T1	**T2**	**<u>Result</u>**
read_lock (Y);		X=50; Y=50
read_item (Y);		Nonserializable because it
unlock (Y);		violated two-phase policy.
	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);		

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

T'1	**T'2**
read_lock (Y);	read_lock (X);
read_item (Y);	read_item (X);
write_lock (X);	write_lock (Y);
unlock (Y);	unlock (X);
read_item (X);	read_item (Y);
X:=X+Y;	Y:=X+Y;
write_item (X);	write_item (Y);
unlock (X);	unlock (Y);

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

Database Concurrency Control

Two-Phase Locking Techniques: The algorithm

Two-phase policy generates two locking algorithms (a) Basic and (b) Conservative.

<u>Conservative</u>: Prevents deadlock by locking all desired data items before transaction begins execution.

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

<u>Strict</u>: 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.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock

$\underline{T'_1}$	$\underline{T'_2}$
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

Deadlock (T'_1 and T'_2)

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

Database Concurrency Control

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.

Database Concurrency Control

Dealing with Deadlock and Starvation

Deadlock detection and resolution

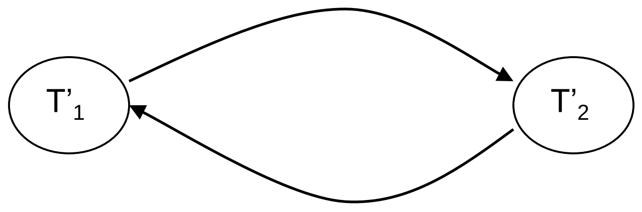
In this approach, deadlocks are allowed to happen. The scheduler maintains a *wait-for-graph* for detecting cycle. If a *cycle* exists, then one transaction involved in the cycle is selected (victim) and rolled-back.

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 transactions of the cycle is selected and rolled back.

Wait-for graph

T'_1	**T'_2**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
(waits for X)	
	write_lock (Y);
	(waits for Y)

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



$$\begin{tikzcd} T'_1 \arrow[r, bend left] & T'_2 \arrow[l, bend left] \end{tikzcd}$$

b) wait-for graph

Database Concurrency Control

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.

Database Concurrency Control

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.

Database Concurrency Control

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.

Timestamps

- 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*.

Database Concurrency Control

Timestamp based concurrency control algorithm

Basic Timestamp Ordering

1. Transaction *T* issues a *write_item*(*X*) operation:
   a. If *read_TS*(*X*) > *TS*(*T*) or if *write_TS*(*X*) > *TS*(*T*), then an younger transaction has already read the data item so abort and roll-back *T* 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 *write_TS*(*X*) > *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 *write_TS*(*X*) ≤ *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*).

Ex: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)				RT = 200	
	r2(A)		RT = 150		
		r3(C)			RT=175
w1(B)				WT=200	
w1(A)			WT=200		
	w2(C)				
	Abort				
		w3(A)			

Why T2 must be aborted (rolled-back)?

Database Concurrency Control

Timestamp based concurrency control algorithm

Strict Timestamp Ordering

1. Transaction *T* issues a *write_item*(*X*) operation:
   a. 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:
   a. If *TS*(*T*) > *write_TS*(*X*), then delay *T* until the transaction *T'* that wrote *X* has terminated (committed or aborted).

Database Concurrency Control

Timestamp based concurrency control algorithm

Thomas's Write Rule

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 the write operation and continue execution. This is because the most recent write 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*).

Database Concurrency Control

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 are satisfied.

Database Concurrency Control

Multiversion technique based on timestamp ordering

Assume X_1, X_2, …, X_n are the versions 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.

Database Concurrency Control

Multiversion technique based on timestamp ordering

To ensure serializability, the following two rules are used.

If transaction T issues *write_item*(X) 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 T; otherwise create a new version X_j and *read_TS*(X_j) = *write_TS*(X_j) = *TS*(T).

If transaction T issues *read_item* (X), 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 T, and set the value of *read _TS*(X_i) to the largest of *TS*(T) and the current *read_TS*(X_i).

Rule 2 guarantees that a read will never be rejected.

Ex: Execution of transactions using multiversion concurrency control

T1	T2	T3	T4	A_0	A_{150}	A_{200}
150	200	175	225			
r1(A)				read		
w1(A)					Create	
	r2(A)				Read	
	w2(A)					Create
		r3(A)			read	
			r4(A)			read

<u>Note</u>: T3 does not have to abort, because it can read an earlier version of A.

Database Concurrency Control

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* where one version must always have been written by some committed transaction. This means a write operation always creates a new version of *X*.

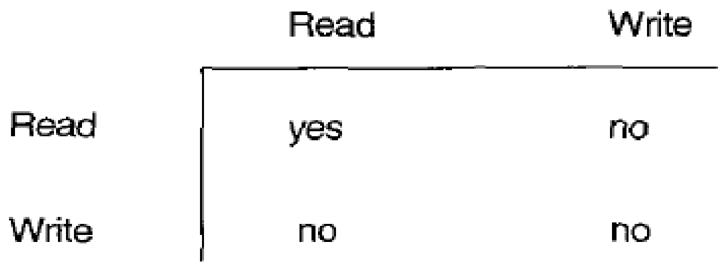
Database Concurrency Control

Multiversion Two-Phase Locking Using Certify Locks

Steps

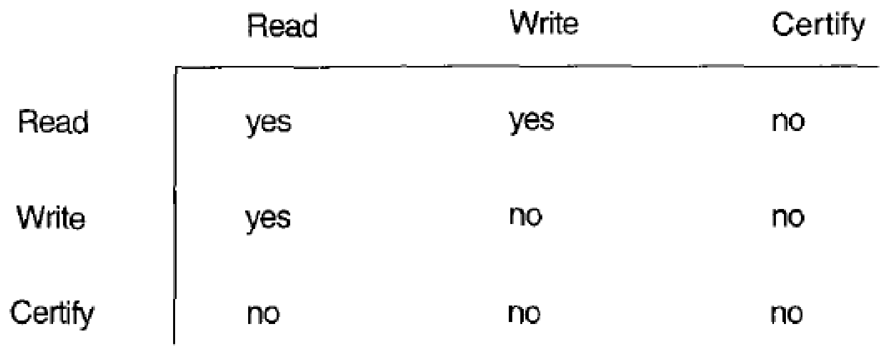
1. *X* is the committed version of a data item.
2. *T* creates a second version *X'* after obtaining a write lock on *X*.
3. Other transactions continue to read *X*.
4. *T* is ready to commit so it obtains a ***certify lock*** on *X'*.
5. The committed version *X* becomes *X'*.
6. *T* releases its ***certify 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	yes	no
Write	yes	no	no
Certify	no	no	no

read/write/certify locking scheme

Database Concurrency Control

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 if upgrading of a read lock to a write lock is allowed.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

In this technique only at the time of commit, serializability is checked and transactions are aborted in case of non-serializable schedules.

Three phases: read phase, validation phase, write phase

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).

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

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.

Database Concurrency Control

Validation (Optimistic) Concurrency Control Schemes

When validating T_i, the first condition is checked first for each transaction T_j, since (1) is the simplest condition to check. If (1) is false then (2) is checked and if (2) is false then (3) is checked. If none of these conditions holds, the validation fails and T_i is aborted.

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

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

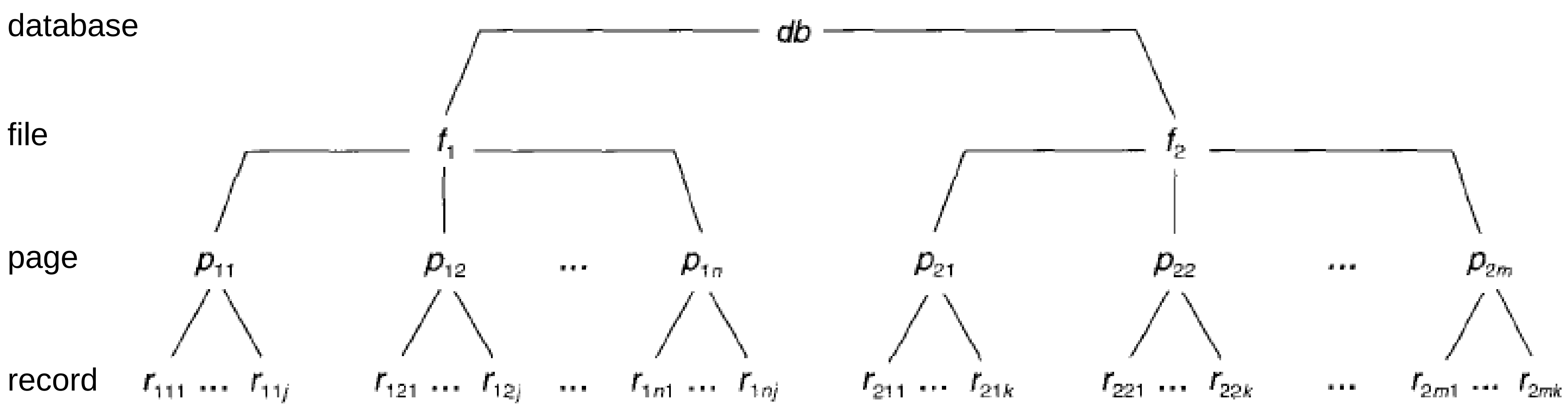
A lockable unit of data defines its ***granularity***. Granularity can be coarse (entire database) or it can be fine (a tuple or an attribute of a relation). Data item granularity significantly affects concurrency control performance. Thus, the degree of concurrency is low for coarse granularity and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

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

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



Database Concurrency Control

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).

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

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

Database Concurrency Control

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 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*.

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution:

T_1 wants to update r_{111}, r_{211}

T_2 wants to update page p_{12}

T_3 wants to read r_{11j} and f_2

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(r_{211})		
Unlock (r_{211})		
Unlock (p_{21})		
Unlock (f_2)		
		S(f_2)

Database Concurrency Control

Granularity of data items and Multiple Granularity Locking

An example of a serializable execution (continued):

T_1	T_2	T_3
	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)