Concurrency control techniques

(Ch. 20, 3rd ed. – Ch. 18, 4th ed., Ch. 18, 5th ed. – Ch. 22, 6th ed. - Ch. 21, 7th ed.)


Concurrency Control
introduction
other topics
granularity
phantoms
interactive transactions
SQL Isolation levels
protocols
locking
what is a lock
binary locks
Shared & exclusive
2PL
basic
conservative
strict
deadlock
prevention
timestamp based
wait-die
wound-wait
waiting based
cautious waiting
waiting
no waiting
detection
wait-for graph
livelock
starvation
timestamps
transactions
database items
read timestamp
write timestamp
algorithm
multiversion
timestamp based
2PL based
optimistic
Multi-granularity

Locking

What is a lock?

A lock is a variable associated with a database item that describes the status of the database item with respect to database operations that can be applied to the database item.

Locks are managed by the Lock Manager within the DBMS

Database items that could be locked vary from a field value up to a whole database:

- field value in a row
- row
- block
- table
- database

See section on granularity

Binary Locks

- A binary lock is in one of two states 0 or 1
  (lock(X) is either 0 or 1)
  values of locks can be held in a *lock table*

- Two lock operations: unlock_item(X) and lock_item(X)
  (these must be implemented as indivisible operations)

- Used to enforce mutual exclusion on data items

- Between lock_item(X) and unlock_item(X), it is said that the transaction holds a lock on item X. When a data item X is locked by a transaction, it cannot be accessed by any other transaction.

Binary Locks: data structures

- lock(X) can have one of two values:

  0 or *1*

  unlocked or *locked*

  etc

- We require a Wait Queue where we keep track of suspended transactions

Lock Table

item	lock	trx_id
X	1	1
Y	1	2

Wait Queue

item	trx_id
X	2
Y	3

Binary Locks: operations

lock_item(X)

- used to gain exclusive access to item X

- if a transaction executes lock_item(X) then
  if lock(X)=0 then
  the lock is granted {lock(X) is set to 1} and
  the transaction can carry on
  {the transaction is said to hold a lock on X}
  otherwise
  the transaction is placed in a wait queue until
  lock_item(X) can be granted
  {i.e. until some other transaction unlocks X}

Binary Locks: operations

unlock_item(X)

- used to relinquish exclusive access to item X
- if a transaction executes unlock_item(X) then
  lock(X) is set to 0
  {note that this may enable some other blocked transaction
  to resume execution}

Binary Locks

Binary locking protocol (rules)

- a lock_item(X) must be issued before any read_item(X) or write_item(X)
- an unlock_item(X) must be issued after all read_item(X) and write_item(X) operations are completed
- a transaction will not issue a lock_item(X) if it already holds a lock on item X
- a transaction will not issue an unlock_item(X) unless it already holds the lock on item X

Example: Binary Locks

time	Transaction1	Transaction2
1	lock_item(X)	
2	read_item(X)	
3		lock_item(X)
4	write_item(X)	
5	unlock_item(X)	
6	commit	
7		read_item(X)
8		unlock_item(X)
9		commit

T2 is placed in the wait queue

T2 can resume

At time 7 the lock_item(X) initiated at time 3 is completed and then the read_item(X) is done

Lock table:

Item	lock	trx_id
x	1	1

Waiting queue:

Item	trx_id
x	2

Lock table:

Item	lock	trx_id
x	1	2

Waiting queue:

Item	trx_id

Shared and Exclusive Locks

Three operations:

- read_lock(X)
- write_lock(X)
- unlock(X)

'read' locks allow more than one transaction to access a data item

Each is an indivisible operation

Use a multiple-mode lock with three possible states

- read-locked — *also called share-locked*
- write-locked — *also called exclusive-locked*
- unlocked

Shared and Exclusive Locks: data structures

- For any data item X, lock(X) can have one of three values: *read-locked*, *write-locked*, *unlocked*
- For any data item X, we need a counter (no_of_readers) to know when all "readers" have relinquished access to X
- We require a Wait Queue where we keep track of suspended transactions

Lock Table

item	lock	no_of_readers	trx_ids
X	1	2	{1, 2}
Y	2	1	2

Wait Queue

item	transaction
X	3

Shared and Exclusive Locks: operations

read_lock(X)

- used to gain shared access to item X
- if a transaction executes read_lock(X) then

  if lock(X) is <u>not</u> “write_locked” then

  the lock is granted

  {lock(X) is set to “read_locked”,

  the “no_of_readers” is incremented by 1},

  and the transaction can carry on

  {the transaction is said to hold a shared lock on X}

  otherwise

  the transaction is placed in a wait queue until

  read_lock(X) can be granted

  {i.e. until some transaction relinquishes exclusive access to X}

Shared and Exclusive Locks: operations

write_lock(X)

- used to gain exclusive access to item X
- if a transaction executes write_lock(X) then

  if lock(X) is "unlocked" then

    the lock is granted {lock(X) is set to "write_locked"}, and the transaction can carry on

    {the transaction is said to hold an exclusive lock on X}

  otherwise

    the transaction is placed in a wait queue until write_lock(X) can be granted

    {i.e. until all other transactions have relinquished their access rights to X - that could be a single "writer" or several "readers"}

Shared and Exclusive Locks: operations

unlock(X)

- used to relinquish access to item X
- if a transaction executes unlock(X) then

    if lock(X) is "read_locked" then

        decrement no_of_readers by 1

        if no_of_readers=0 then set lock(X) to "unlocked"

    otherwise

        set lock(X) to "unlocked"

{note that setting lock(X) to "unlocked" may enable a blocked transaction to resume execution}

Example: Shared and Exclusive Locks

Time	Transaction1	Transaction2
1	read_lock(X)	
2	read_item(X)	
3		read_lock(X)
4		read_item(X)
5		read_lock(Y)
6		read_item(Y)
	write_lock(Y)	
7		write_lock(Z)
8		read_item(Z)
9		Z := X + Y + Z
10	waiting	write_item(Z)
12		unlock(X)
13		unlock(Y)
14		unlock(Z)
15	read_item(Y)	
16	Y := X + Y	
17	write_item(Y)	

What are the contents of the wait queue and lock table at times 1 through 15?

The write_lock at time 7 causes T1 to be suspended, and it isn't until time 15 that the write_lock actually completes

Lock table:

Item	lock	no-of-readers	Trx_ids
X	1	1	{1}

Waiting queue:

Item	Trx_id

Lock table:

Item	lock	no-of-readers	Trx_ids
X	1	1	{1}

Waiting queue:

Item	Trx_id

Lock table:

Item	lock	no-of-readers	Trx_ids
X	1	2	{1, 2}

Waiting queue:

Item	Trx_id

Lock table:

Item	lock	no-of-readers	Trx_ids
X	1	2	{1, 2}
Y	1	1	{2}
Z	2	1	{2}

Waiting queue:

Item	Trx_id
Y	1

Shared and Exclusive Locks

locking protocol (rules); a transaction T

- must issue read_lock(X) or write_lock(X) before read-item(X)
- must issue write_lock(X) before write-item(X)
- must issue unlock(X) after all read_item(X) and write_item(X) operations are completed
- will not issue a read_lock(X) if it already holds a read or write lock on X (*can be relaxed, to be discussed*)
- will not issue a write_lock(X) if it already holds a read or write lock on X (*can be relaxed, to be discussed*)
- will not issue an unlock unless it already holds a read lock or a write lock on X

Shared and Exclusive Locks

Figure 18.3 (a)

T1	T2
read_lock(Y)	read_lock(X)
read_item(Y)	read_item(X)
unlock(Y)	unlock(X)
write_lock(X)	write_lock(Y)
read_item(X)	read_item(Y)
X:=X+Y	Y:=X+Y
write_item(X)	write_item(Y)
unlock(X)	unlock(Y)

*If initial values of X and Y are 20 and 30 respectively, then **correct** values of X and Y after T1 and T2 execute will be either 50 and 80, or 70 and 50 respectively*

T1 → T2	T2 → T1
T1: X = 20	T2: X = 20
Y = 30	Y = 30
X := 20 + 30	Y := 20 + 30
X = 50	Y = 50
T2: X = 50	T1: X = 20
Y = 30	Y = 50
Y := 50 + 30	X := 20 + 50
Y = 80	X = 70

Shared and Exclusive Locks

T1	T2	T1	T2
read_lock(Y)			
read_item(Y)		p1_Y = 30	
unlock(Y)			
	read_lock(X)		
	read_item(X)		p2_X = 20
	unlock(X)		
	write_lock(Y)		
	read_item(Y)		p2_Y = 30
	Y:=X+Y		p2_Y = 50
	write_item(Y)		d_Y = 50
	unlock(Y)		
write_lock(X)			
read_item(X)		p1_X = 20	
X:=X+Y			
write_item(X)		p1_X = 50	
unlock(X)		d_X = 50	

Result is X=50 and Y=50, which is ***incorrect***

Shared and Exclusive Locks (2PL)

Conversion of Locks

Recall a transaction T

- will not issue a read_lock(X) if it already holds a read or write lock on X

  Can permit a transaction to ***downgrade*** a lock from a write to a read lock

- will not issue a write lock(X) if it already holds a read or write lock on X

  Can permit a transaction to ***upgrade*** a lock on X from a read to a write lock if no other transactions hold a read lock on X

Shared and Exclusive Locks (2PL)

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 operations in the transaction.

- previous protocols do not guarantee serializability
- Serializability is guaranteed if we enforce the two-phase locking protocol:

  all locks must be acquired before any locks are relinquished

- transactions will have a *growing* and a *shrinking* phase
- any upgrading of locks must occur in the growing phase
- any downgrading of locks must occur in the shrinking phase

Shared and Exclusive Locks (2PL)

Figure 18.4

T1'	T2'
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)

These transactions obey the 2PL protocol

Shared and Exclusive Locks (2PL)

T1'	T2'
read_lock(Y) read_item(Y) write_lock(X)	
	read_lock(X)
unlock(Y) read_item(X) X:=X+Y write_item(X) unlock(X)	
	read_item(X) write_lock(Y) unlock(X) read_item(Y) Y:=X+Y write_item(Y) unlock(Y)

The read_lock causes T2 to be suspended, and it isn't until later, just prior to read_item(X), that the read_lock actually completes

With 2PL these transactions will be run correctly

Shared and Exclusive Locks (2PL)

T1'	T2'	T2'
read_lock(Y)	**read_lock(Z)**	**read_lock**(X)
read_item(Y)	read_item(Z)	read_item(X)
write_lock(X)	**write_lock**(Y)	**write_lock**(Y)
unlock(Y)	**unlock**(Z)	**unlock**(X)
read_item(X)	read_item(Y)	read_item(Y)
X:=X+Y	Y:=Z+Y	Y:=X+Y
write_item(X)	write_item(Y)	write_item(Y)
unlock(X)	**unlock**(Y)	**unlock**(Y)

These transactions obey the 2PL protocol

T1'	T2'
read_lock(Y) read_item(Y) write_lock(X)	
	read_lock(Z) read_item(Z) write_lock(Y)
unlock(Y) read_item(X) X:=X+Y write_item(X) unlock(X)	
	unlock(Z) read_item(Y) Y:=Z+Y write_item(Y) unlock(Y)

- The 2PL can produce a deadlock.

T1’		T2’
read_lock(Y)		
read_item(Y)		
	wait	read_lock(X)
		read_item(X)
		write_lock(Y)
write_lock(X)	*wait*	
unlock(Y)		
read_item(X)		
X:=X+Y		
write_item(X)		
unlock(X)		
		unlock(X)
		read_item(Y)
		Y:=X+Y
		write_item(Y)
		unlock(Y)

Variations on 2PL

Basic 2PL

- previous protocol

Conservative 2PL

- transactions must lock all items prior to the transaction executing
- if any lock is not available then none are acquired - all must be available before execution can start
- free of deadlocks

Strict 2PL

- a transaction does not release any write-locks until after it commits or aborts
- **most popular** of these schemes
- recall strict schedule avoids cascading rollback
- undoing a transaction can be efficiently conducted.

Shared and Exclusive Locks (Strict 2PL)

Figure 18.4

T1'	T2'
read_lock(Y)	read_lock(X)
read_item(Y)	read_item(X)
write_lock(X)	write_lock(Y)
read_item(X)	read_item(Y)
X:=X+Y	Y:=X+Y
write_item(X)	write_item(Y)
commit	**commit**
unlock(Y)	**unlock(X)**
unlock(X)	**unlock(Y)**

These transactions obey the Strict 2PL protocol

Shared and Exclusive Locks (Strict 2PL)

Figure 18.4

T1’	T1’	T2’	T2’
read_lock(Y)	read_lock(Y)	read_lock(X)	read_lock(X)
read_item(Y)	read_item(Y)	read_item(X)	read_item(X)
write_lock(X)	write_lock(X)	write_lock(Y)	write_lock(Y)
read_item(X)	read_item(X)	read_item(Y)	read_item(Y)
X:=X+Y	X:=X+Y	Y:=X+Y	Y:=X+Y
write_item(X)	write_item(X)	write_item(Y)	write_item(Y)
unlock(Y)	**commit**	**commit**	**unlock(X)**
commit	**unlock(Y)**	**unlock(X)**	**commit**
unlock(X)	**unlock(X)**	**unlock(Y)**	**unlock(Y)**

These transactions obey the Strict 2PL protocol

Shared and Exclusive Locks (Strict 2PL)

T1'	T2'
read_lock(Y) read_item(Y) write_lock(X)	
	read_lock(X)
read_item(X) X:=X+Y write_item(X) **commit** **unlock(Y), unlock(X)**	
	read_item(X) write_lock(Y) read_item(Y) Y:=X+Y write_item(Y) **commit** **unlock(X), unlock(Y)**

Shared and Exclusive Locks (Strict 2PL)

T1'	**T2'**
read_lock(Y)	read_lock(Z)
read_item(Y)	read_item(Z)
write_lock(X)	write_lock(Y)
read_item(X)	read_item(Y)
X:=X+Y	Y:=Z+Y
write_item(X)	write_item(Y)
commit	**commit**
unlock(Y)	**unlock(Y)**
unlock(X)	**unlock(Z)**

These transactions obey the Strict 2PL protocol

Shared and Exclusive Locks (Strict 2PL)

T1'	T1'	T2'	T2'
read_lock(Y)	read_lock(Y)	read_lock(Z)	read_lock(X)
read_item(Y)	read_item(Y)	read_item(Z)	read_item(X)
write_lock(X)	write_lock(X)	write_lock(Y)	write_lock(Y)
read_item(X)	read_item(X)	read_item(Y)	read_item(Y)
X:=X+Y	X:=X+Y	Y:=Z+Y	Y:=X+Y
write_item(X)	write_item(X)	write_item(Y)	write_item(Y)
unlock(Y)	**commit**	**commit**	**unlock(X)**
commit	**unlock(Y)**	**unlock(Y)**	**commit**
unlock(X)	**unlock(X)**	**unlock(Z)**	**unlock(Y)**

Shared and Exclusive Locks (Strict 2PL)

T1'	T2'
read_lock(Y) read_item(Y)	
	read_lock(Z) read_item(Z) write_item(Y)
write_lock(X) read_item(X) X:=X+Y write_item(X) **commit** **unlock(Y), unlock(X)**	
	read_item(Y) Y:=Z+Y write_item(Z) **commit** **unlock(Y), unlock(Z)**

Shared and Exclusive Locks (Strict 2PL)

T1’	T2’
read_lock(Y) read_item(Y)	
	read_lock(X) read_item(X)
write_lock(X)	
	write_lock(Y)
deadlock	
read_item(X) X:=X+Y write_item(X) **commit** **unlock(Y), unlock(X)**	read_item(Y) Y:=X+Y write_item(Y) **commit** **unlock(X), unlock(Y)**

Deadlock

Deadlock occurs when two or more transactions are in a simultaneous wait state, each one waiting for one of the others to release a lock.

T1	**T2**
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
waiting	
	write_lock(Y)
	waiting

deadlock

Deadlock Prevention

1. Conservative 2PL
2. Always locking in a predefined sequence
3. **Timestamp based**
4. **Waiting based**
5. Timeout based

Deadlock Prevention - Timestamp based

- Each transaction is assigned a timestamp (TS).

  If a transaction T1 starts before transaction T2,

  then TS(T1) < TS(T2); T1 is *older* than T2.

- Two schemes:

  Wait-die

  Wound-wait

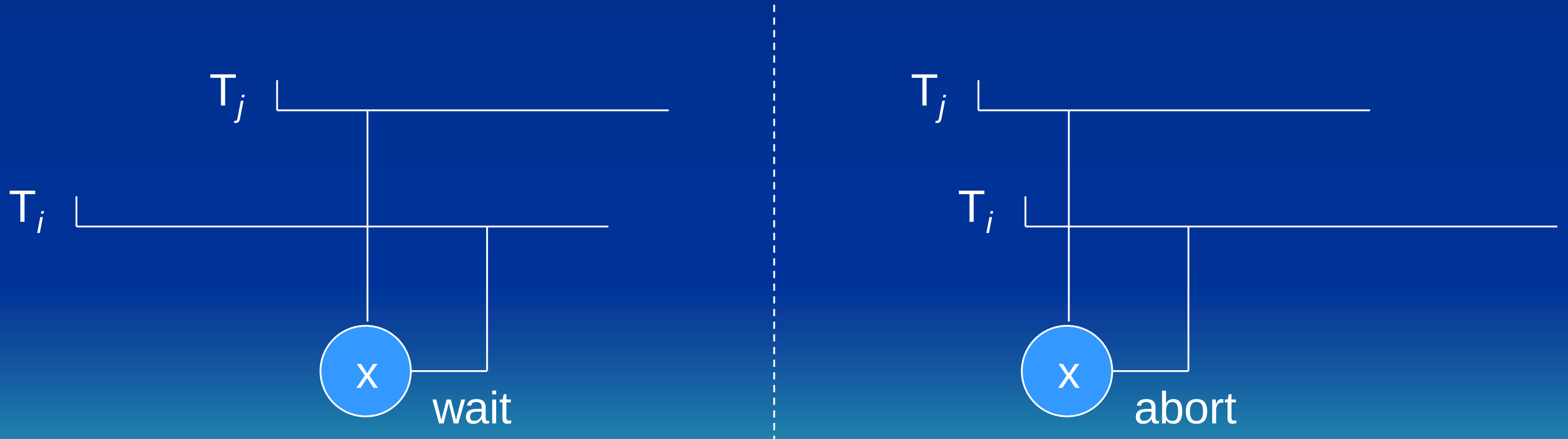
- Both schemes will cause aborts even though deadlock would not have occurred.

Deadlock Prevention: Wait-die

Suppose Ti tries to lock an item locked by Tj.

If Ti is the older transaction then Ti will wait.

Otherwise, Ti is aborted and restarts later with the same timestamp.



Example: Wait-die

T1 starts before T2 so T1 is older

T1	**T2**
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
	write_lock(Y)
	abort

T1 is older and so it is allowed to wait.

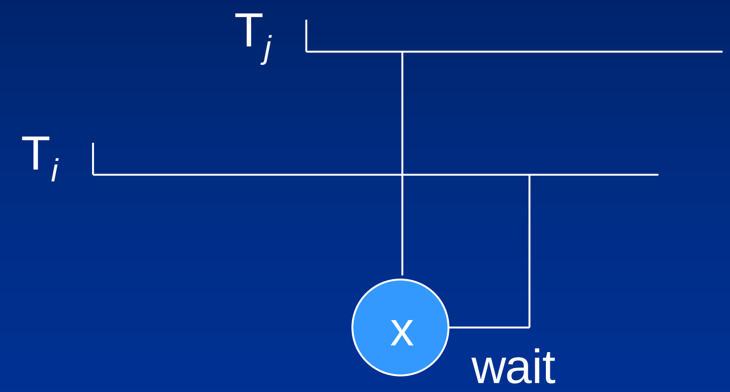
T2 is younger and so it is aborted, which results in its locks being released, and that allows T1 to carry on:

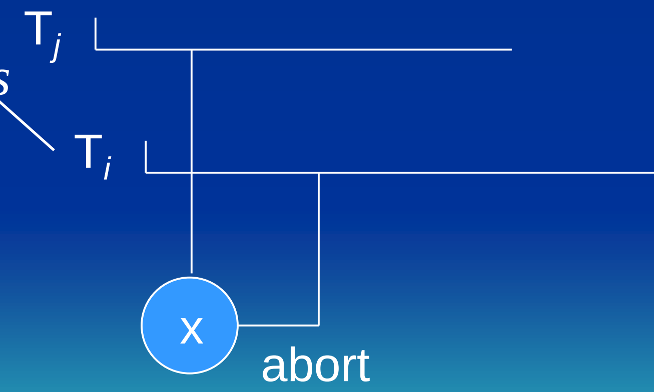
abort → *can resume*

Example: Wait-die

T1 starts before T2
so T1 is older

T1	T2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
T1 is older and so it is allowed to wait.	
	write_lock(Y)
	T2 is younger and so it is aborted, which results in its locks being released, and that allows T1 to carry on:
	abort
can resume	





Deadlock Prevention: Wound-wait

Suppose Ti tries to lock an item locked by Tj.

If Ti is the older transaction

then Tj is aborted and restarts later with the same timestamp;

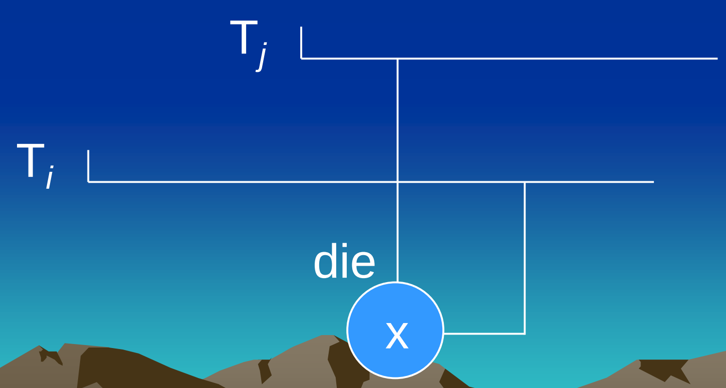
otherwise Ti is allowed to wait.

Example: Wound-wait

T1 starts before T2
so T1 is older

T1	T2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
	aborted

T1 is older, so T2 is aborted and that allows T1 to carry on.



Example: Wound-wait

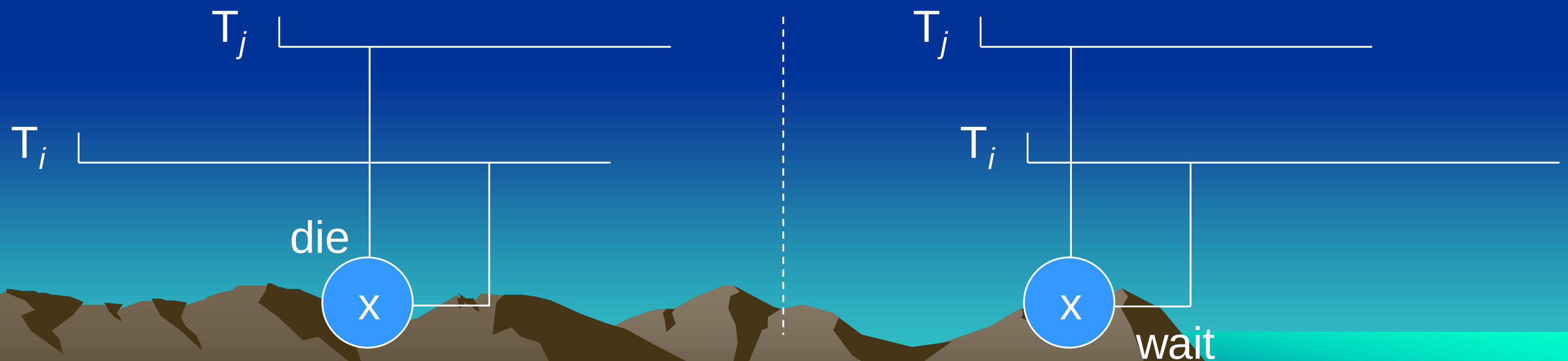
T2 starts before T1
so T2 is older

T1	**T2**
	read_lock(X)
	read_item(X)
read_lock(Y)	
read_item(Y)	
write_lock(X)	
T1 is younger, so wait.	write_lock(Y)
Aborted since T2 is older.	

Example: Wound-wait

T2 starts before T1
so T2 is older

T1	**T2**
	read_lock(X)
	read_item(X)
read_lock(Y)	
read_item(Y)	
write_lock(X)	
T1 is younger, so wait.	write_lock(Y)
Aborted since T2 is older.	



Deadlock Prevention - Waiting based

- No timestamps
- Two schemes:

  no waiting

  cautious waiting
- Both schemes will cause aborts even though deadlock would not have occurred.

Deadlock Prevention: No waiting

Suppose Ti tries to lock an item locked by Tj.

If Ti is unable to get the lock
then Ti is aborted and restarted after some time delay.

Transactions may be aborted and restarted needlessly.

Example: No waiting

T1	T2
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
T1 is blocked and aborted:	
abort	
	write_lock(Y)

since T1 was aborted, T2 gets the lock and is able to carry on.

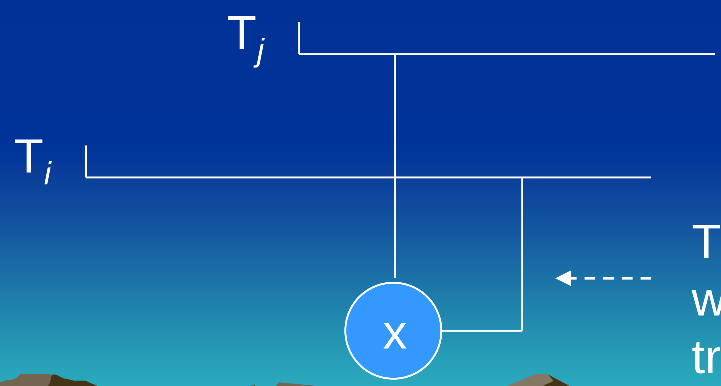
Deadlock Prevention: Cautious waiting

Suppose Ti tries to lock an item locked by Tj.

If Tj is not waiting on another transaction,

then Ti is allowed to wait;

otherwise Ti is aborted.



T_i waits or aborts depending on whether T_j is waiting for some other transaction or not.

Example: Cautious waiting

T1	T2
read_lock(Y) read_item(Y)	
	read_lock(X) read_item(X)
write_lock(X) *T1 is allowed to wait since T2 is not blocked.*	
	write_lock(Y) *T2 is aborted since it is blocked by a transaction that is also blocked.*
	abort *Now, T1 can resume.*
carries on	

Deadlock Detection

Periodically check for deadlock in the system.

Detection algorithm uses a ***wait-for graph****:*

- *one node for each transaction*
- *an edge (Ti → Tj) is created if Ti is waiting for Tj to release a lock (the edge is removed when Tj releases the lock and Ti is then unblocked).*
- *if the graph has a cycle then there is deadlock.*
- *if there is deadlock then a victim is chosen and it is aborted.*

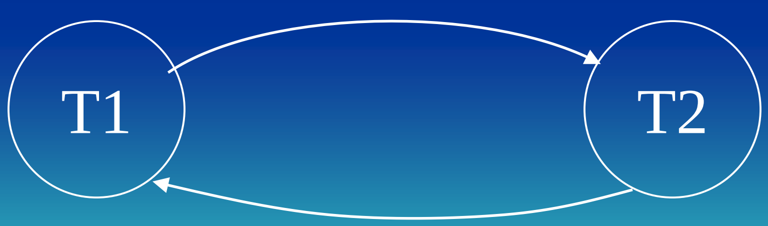
Example: Deadlock Detection

Figure 18.5

T1	**T2**
read_lock(Y)	
read_item(Y)	
	read_lock(X)
	read_item(X)
write_lock(X)	
waiting	
	write_lock(Y)
	waiting

Wait-for graph:

has a cycle!



Starvation

If a transaction is continually restarted and then aborted, it is in a state of starvation.

Livelock

If a transaction is continually waiting for a lock, it is in a state of Livelock.

Concurrency Control - Timestamps

- Each transaction is assigned a timestamp (TS)

  If a transaction T1 starts before transaction T2,

  then TS(T1) < TS(T2);   T1 is *older* than T2.

- Whereas locking-based methods synchronize transaction execution so that the interleaved execution is equivalent to ***some*** serial schedule, timestamping-based methods synchronize transaction execution so that the interleaved execution is equivalent to a ***specific*** serial execution - namely, that defined by the chronological order of the transaction timestamps.

Consider four transactions: T1, T2, T3, T4.

Assume that TS(T1) < TS(T2) < TS(T3) < TS(T4).

We may have 4! = 24 different serial execution of these transactions. Each of them is considered correct:

$$T1 \Rightarrow T2 \Rightarrow T3 \Rightarrow T4$$
$$T2 \Rightarrow T1 \Rightarrow T3 \Rightarrow T4$$
… …
$$T4 \Rightarrow T3 \Rightarrow T2 \Rightarrow T1$$

But the method based on ‘timestamps’ synchronizes the interleaved execution of transactions so that it is equivalent to a specific serial execution:

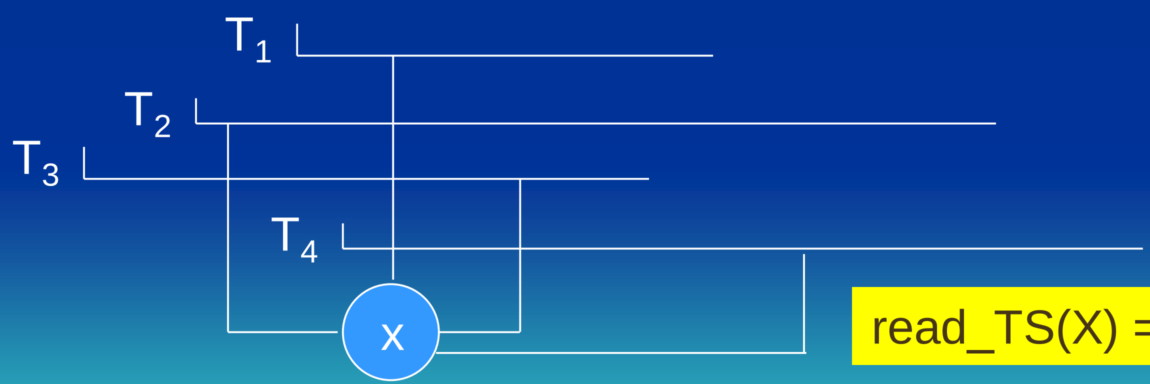
$$T1 \Rightarrow T2 \Rightarrow T3 \Rightarrow T4$$

Database Item Timestamps

- Each database item X has 2 timestamps:
  - the **read timestamp** of X, read_TS(X), is the largest timestamp among all transaction timestamps that have successfully read X.
  - the **write timestamp** of X, write_TS(X), is the largest timestamp among all transaction timestamps that have successfully written X.

Database Item Timestamps

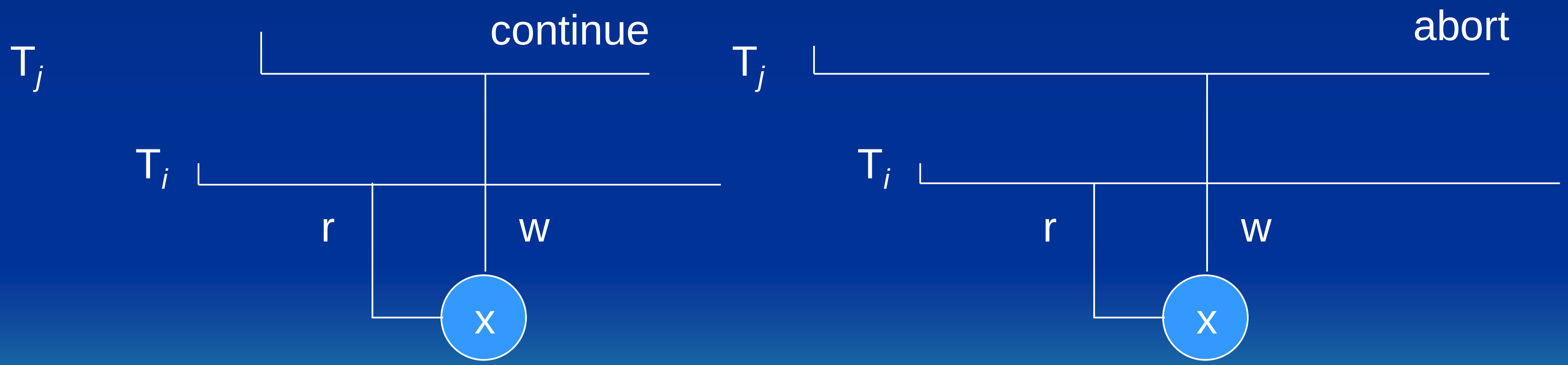
- Each database item X has 2 timestamps:
  - the **read timestamp** of X, read_TS(X), is the largest timestamp among all transaction timestamps that have successfully read X.
  - the **write timestamp** of X, write_TS(X), is the largest timestamp among all transaction timestamps that have successfully written X.



read_TS(X) = timestamp of T_4

Timestamp Ordering (TO) Algorithm

- When a transaction T tries to read or write an item X, the timestamp of T is compared to the read and write timestamps of X to ensure the timestamp order of execution is not violated.
- If the timestamp order of execution is violated, then T is aborted and resubmitted later with a new timestamp.



read_TS(X) = timestamp of T_i

- Deadlock will not occur.
- Cyclic restart of a transaction can occur.
- Cascading rollback can occur.

Timestamp Ordering (TO) Algorithm - in detail

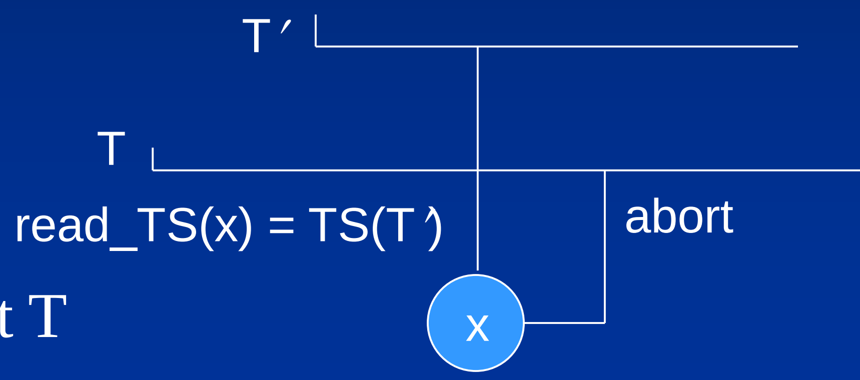
- If T issues **write_item**(X) then

  if {read_TS(X) > TS(T) or write_TS(X) > TS(T)} then abort T

  otherwise (*TS(T) ≥ read_TS(X) and TS(T) ≥ write_TS(X)*)

  execute write_item(X)

  set write_TS(X) to TS(T)



- if T issues **read_item**(X) then

  if write_TS(X) > TS(T) then abort T

- otherwise (*TS(T) ≥ write_TS(X)*)

  execute read_item(X)

  set read_TS(X) to max{TS(T), read_TS(X)}

Example: TO

	T1	T2
TS	5	10

Initially, the timestamps for all the data items are set to 0.

Time	T1	T2
1	read_item(Y)	
2		read_item(X)
3	write_item(X)	
	aborted	
4		write_item(Y)
5		commit
6	*could be restarted*	

What is the schedule for T1 and T2? Assuming all initial data item timestamps are 0, what are the various read and write timestamps?

Example: TO

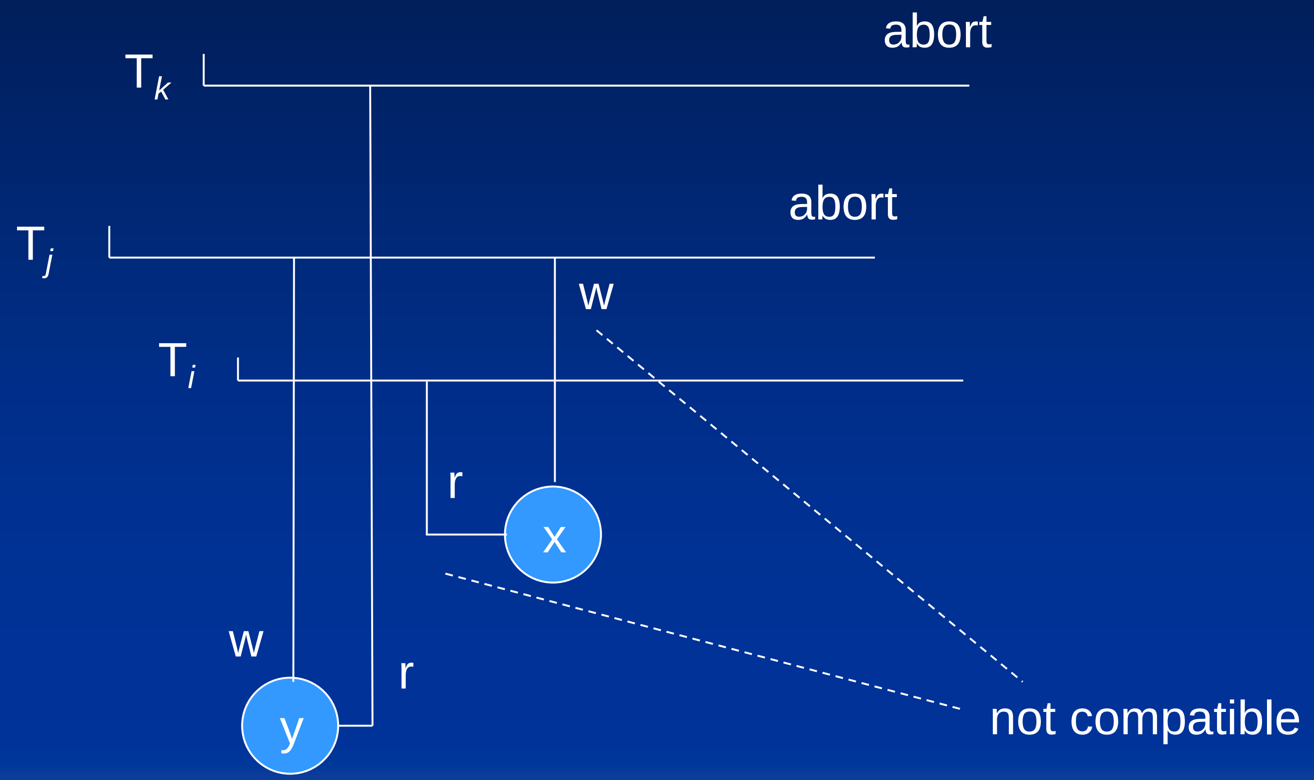
	T1	T2
TS	5	10

Initially, the timestamps for all the data items are set to 0.

Time	T1	T2
1	read_item(Y)	
2	→ read_TS(Y) = 5	read_item(X) → read_TS(X) = 10
3	write_item(X) ←- - - - - - - - -	read_TS(x) = 10 > TS(T1) = 5
	aborted	
4		write_item(Y) → read_TS(Y) = 10
5		commit
6	*could be restarted*	

What is the schedule for T1 and T2? Assuming all initial data item timestamps are 0, what are the various read and write timestamps?

Why does the cascading rollback can occur?



The abortion of T_j leads to the abortion of T_k.

Concurrency Control - Multiversion 2PL

- Basic idea is to keep older version of data items around.
- When a transaction requires access to an item, an appropriate version is chosen to maintain serializability, if possible.
- Some read operations that would be rejected by other techniques can still be accepted by reading an older version of an item.

- No cascading rollback.
- Deadlock can occur.
- In general, requires more storage.
- Particularly adaptable to temporal databases.

Concurrency Control - Multiversion 2PL

- Two versions of data items
- Three locking modes: read, write, certify
- Certify lock is issued before a transaction’s commit on all those data items which are currently write-locked by itself.
- Avoids cascading rollback

Concurrency Control - Multiversion 2PL

- **lock compatibility table:**

Type of lock already held by another transaction

	read	*write*	*certify*
read	yes	yes	no
write	yes	no	no
certify	no	no	no

Type of lock requested

‘yes’ means that the lock can be granted

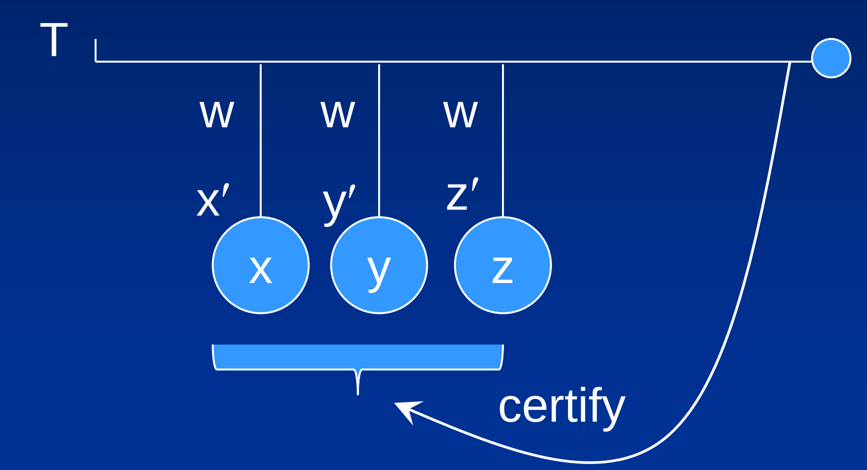
‘no’ means the lock cannot be granted and so the transaction will have to wait

Concurrency Control - Multiversion 2PL

held by another transaction

	read	*write*	*certify*
read	yes	yes	no
write	yes	no	no
certify	no	no	no

Not allowed under standard 2PL, but is allowed here …



Concurrency Control - Multiversion 2PL

Protocol (two-version 2PL):

- Write_item(X)
  - creates a new version of X, X', for the updating transaction
  - committed version of X is still around for other transactions to read
- Commit
  - Before it can commit, T must obtain certify locks on all items that it currently holds write locks on.
  - If the transaction can commit, the committed value of any updated record, X, is set to the value of X', and X' is discarded.
- Certify_item(X)
  - set certify lock on X
  - may be delayed while other transactions hold read locks on X

Concurrency Control - Multiversion 2PL

- Read_item(X)
  - a read obtains the committed value of X.
- Abort

By the multiversion 2PL, we will definitely have no cascading rollback.

Example: Multiversion 2PL

Time	T1	T2
0		sum:=0
1	read_lock(X)	
2	read_item(X)	
3	X:=X-N	
4	write_lock(X)	
5	write_item(X)	
6		read_lock(X)
7		read_item(X)
8		sum:=sum+X
9		read_lock(Y)
10		read_item(Y)
11		sum:=sum+Y
12	read_lock(Y)	
13	read_item(Y)	
14	Y=Y+N +1	
15	write_lock(Y)	
16	write_item(Y)	

T1 creates X'

T2 reads X

X = 20, Y = 30, N = 10

Time 0: d_X = 20, d_Y = 30

Time 5: d_X = 20, d_Y = 30

d_X′ = d_X – N

= 20 – 10

= 10

Time 11: d_X = 20, d_Y = 30

d_X′ = 10

sum = d_X + d_Y

= 20 + 30 = 50

Time 16: d_X = 20, d_Y = 30

d_X′ = 10, d_Y′ = 30 + 10 + 1= 41

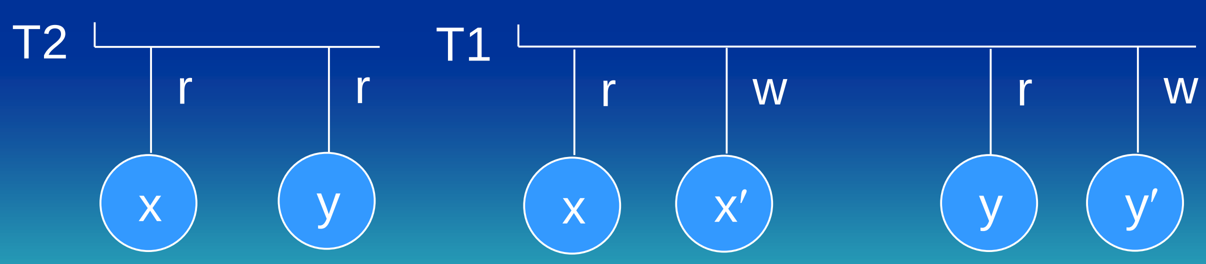
Example: Multiversion 2PL

Time	T1	T2
17	certify(X, Y)	
18		unlock(X)
19		unlock(Y)
20		commit
21	unlock(X)	
22	unlock(Y)	
23	commit	

after commit of T1

d_X = 10, d_Y = 41

$T2 \rightarrow T1$



Concurrency Control - Optimistic

- No checking for interference is done while a transaction is executing
- transactions operate on their own local copies of data items
- when a transaction executes commit, i.e. it is ending, the transaction enters a *validation* phase where serializability is checked
- Reduces overhead
- Useful if there is little interference between transactions

Concurrency Control - Optimistic

- a transaction has three phases
  - *read* - reads operate on database; writes operate on local copies
  - *validation* - check for serializability
  - *write* - if serializability test is satisfied, the database is updated otherwise the transaction is aborted

- read set
- write set

Concurrency Control - Optimistic

- Validation phase:

suppose Ti is in its validation phase, and Tj is any transaction that has committed or is also in its validation phase, then one of 3 conditions must be true for serializability to hold:

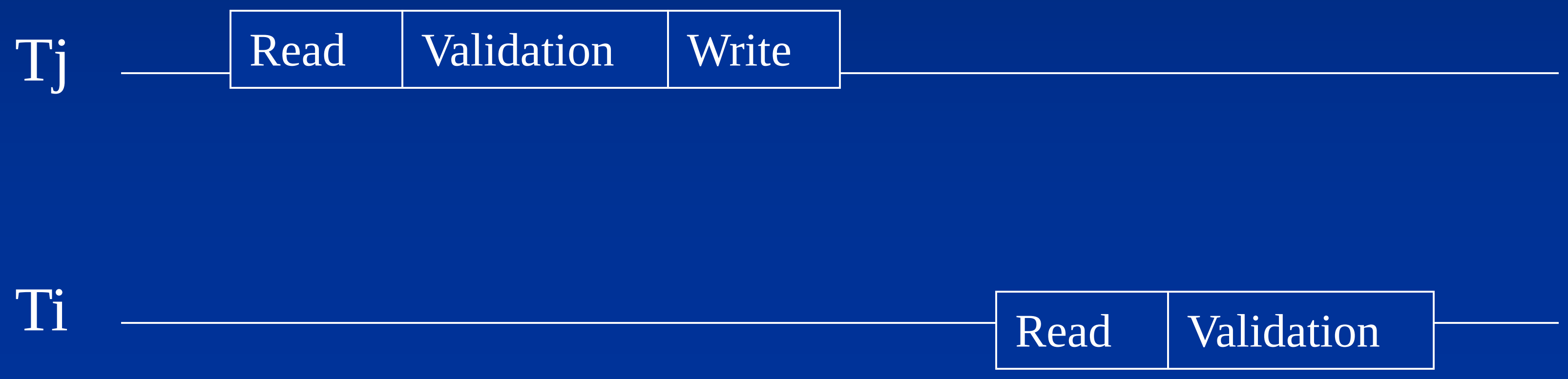
1. Tj completes its write phase before Ti starts its read phase
2. Ti starts its write phase after Tj completes its write phase, and the read set of Ti has no items in common with the write set of Tj
3. both the read set and write set of Ti have no items in common with the write set of Tj, and Tj completes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

If none of these conditions hold, Ti is aborted

Concurrency Control - Optimistic

- Condition 1:

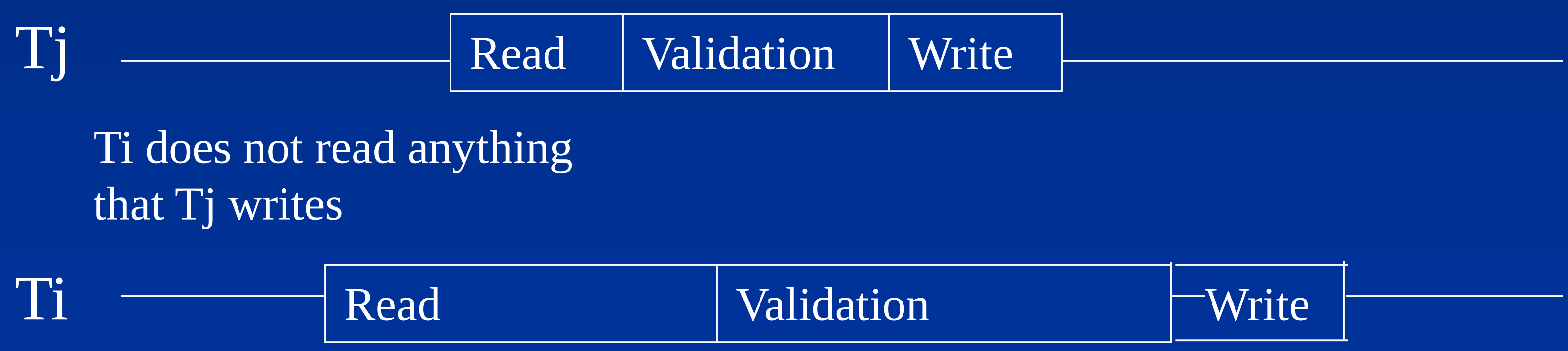
  Tj completes its write phase before Ti starts its read phase



Concurrency Control - Optimistic

- Condition 2:

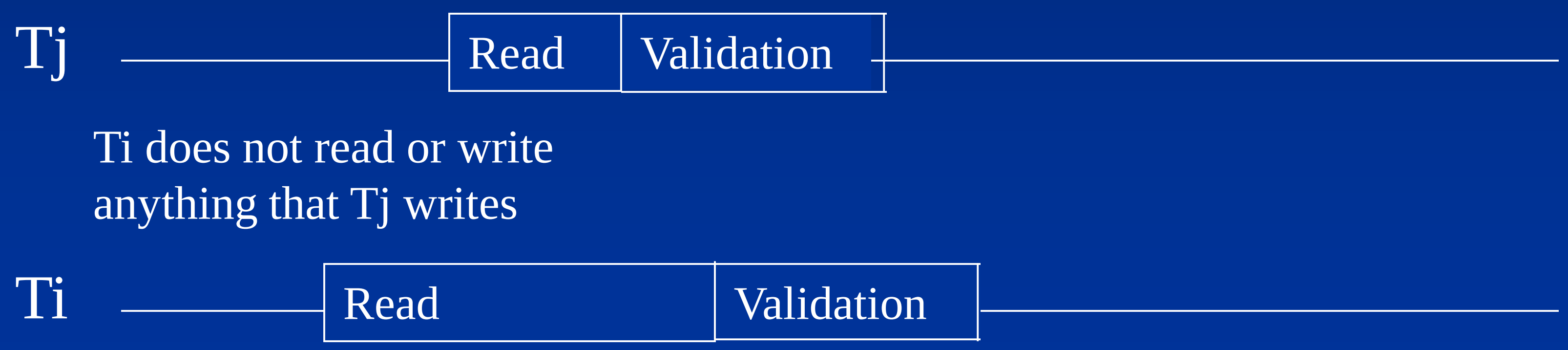
  Ti starts its write phase after Tj completes its write phase, and the read set of Ti has no items in common with the write set of Tj



Concurrency Control - Optimistic

- Condition 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



Granularity of Data Items and Multiple Granularity Locking

- Database is formed of a number of named data items.
- Data item:
  a field value of a database record
  a database record
  a disk block
  a whole table
  a whole file
  a whole database
- The size of data item is often called the data item granularity.
  fine granularity - small data size
  coarse granularity - large data size

Granularity of Data Items and Multiple Granularity Locking

- The larger the data item size is, the lower the degree of concurrency.
- The smaller the data size is, the more the number of items in the database.

  - A larger number of active locks will be handled by the lock manager, and more lock and unlock operations will be performed, causing a higher system overhead.
  - More storage space will be required for storing the lock table.

What is the best item size?

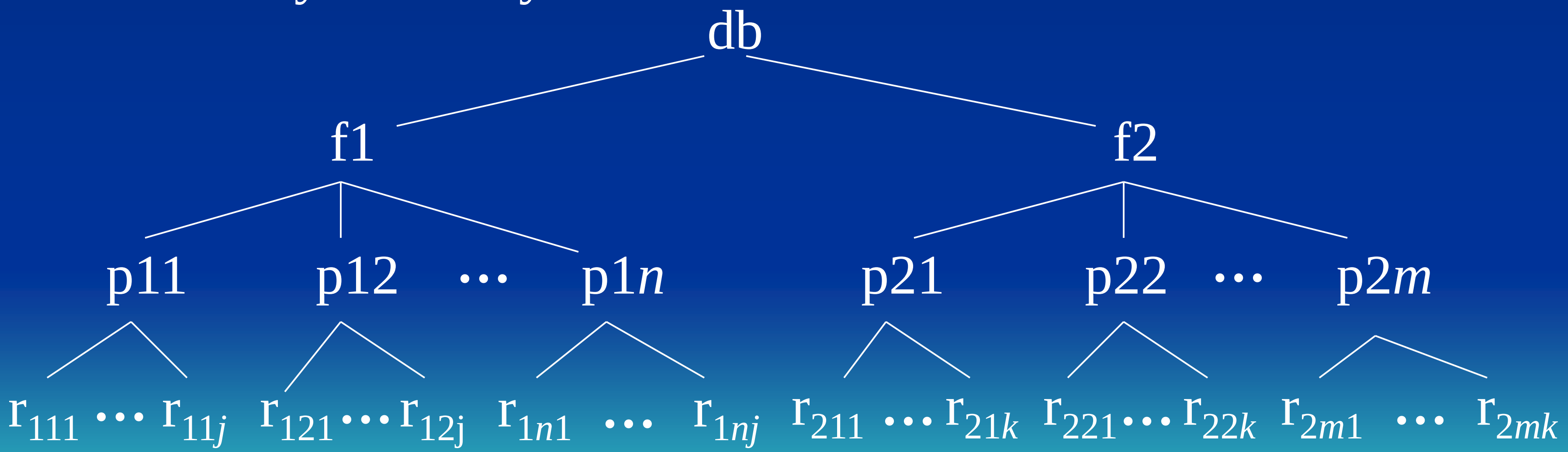
Answer: it depends on the types of transactions involved.

Granularity of Data Items and Multiple Granularity Locking

- Multiple granularity level locking

  Since the best granularity size depends on the given transaction, it seems appropriate that a database system supports multiple levels of granularity, where the granularity level can be different for various mixes of transactions.

  Granularity hierarchy:



Granularity of Data Items and Multiple Granularity Locking

- Problem with only shared and exclusive locks

  T1: updates all the records in file f1.

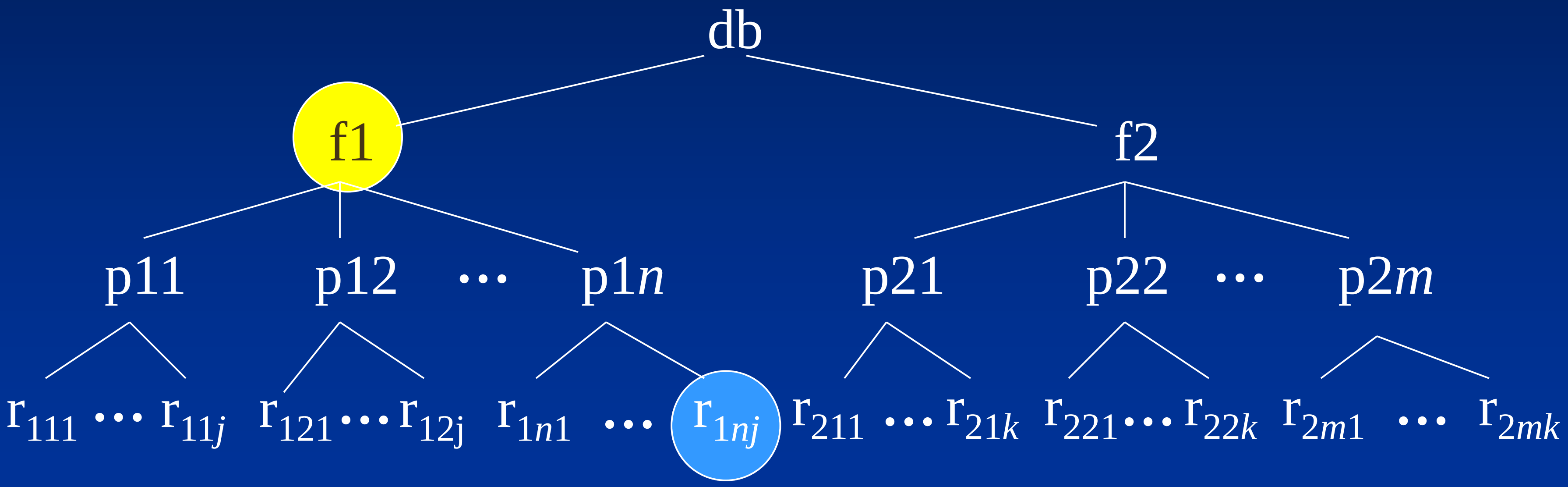
  T2: read record r_{1nj}.

  Assume that T1 comes before T2:

  - T1 locks f1.
  - Before T2 is executed, the compatibility of the lock on r_{1nj} with the lock on f1 should be checked.
  - This can be done by traversing the granularity hierarchy bottom-up (from leaf r_{1nj} to *p1n to* db).

  Assume that T2 comes before T1:

  - T2 locks r_{1nj}.
  - Before T1 is executed, the compatibility of the lock on f1 with the lock on r_{1nj} should be checked.
  - It is quite difficult for the lock manager to check all nodes below f1.


db
f1
f2
p11
p12
…
p1n
p21
p22
…
p2m
r_{111} … r_{11j}
r_{121} … r_{12j}
r_{1n1} … r_{1nj}
r_{211} … r_{21k}
r_{221} … r_{22k}
r_{2m1} … r_{2mk}

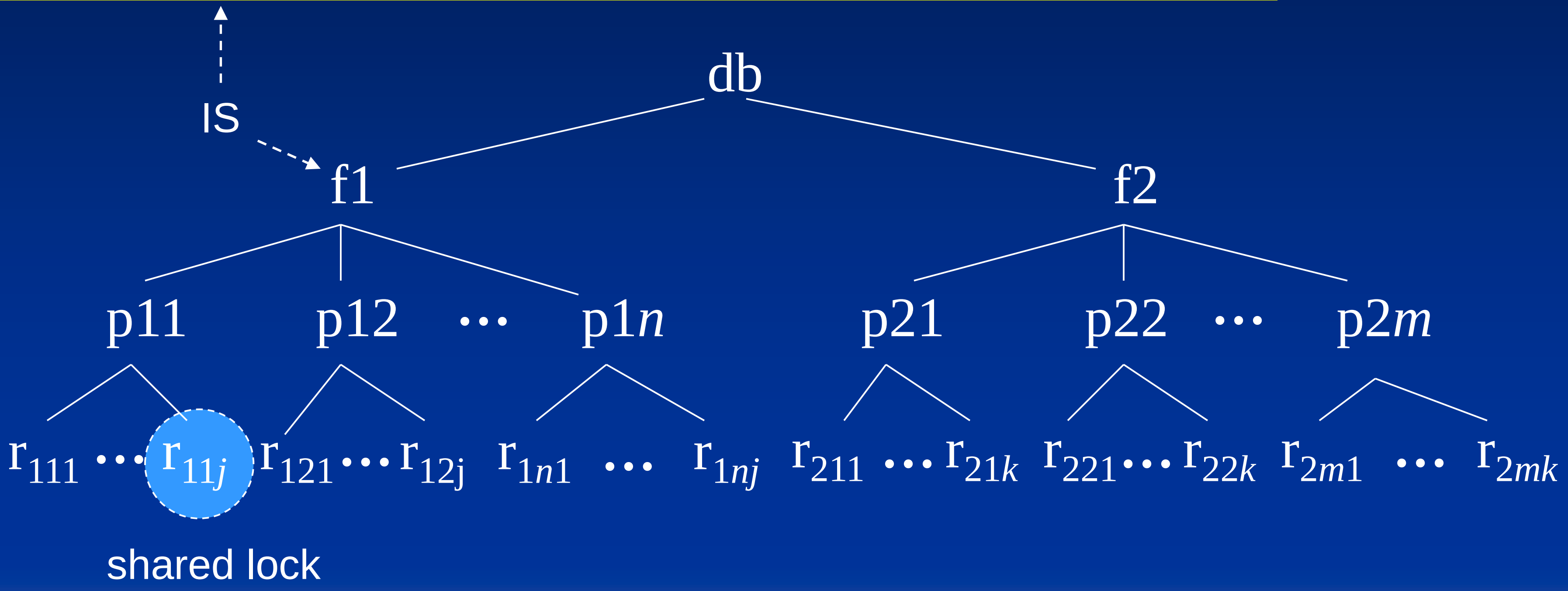
Granularity of Data Items and Multiple Granularity Locking

- Solution: intention locks.

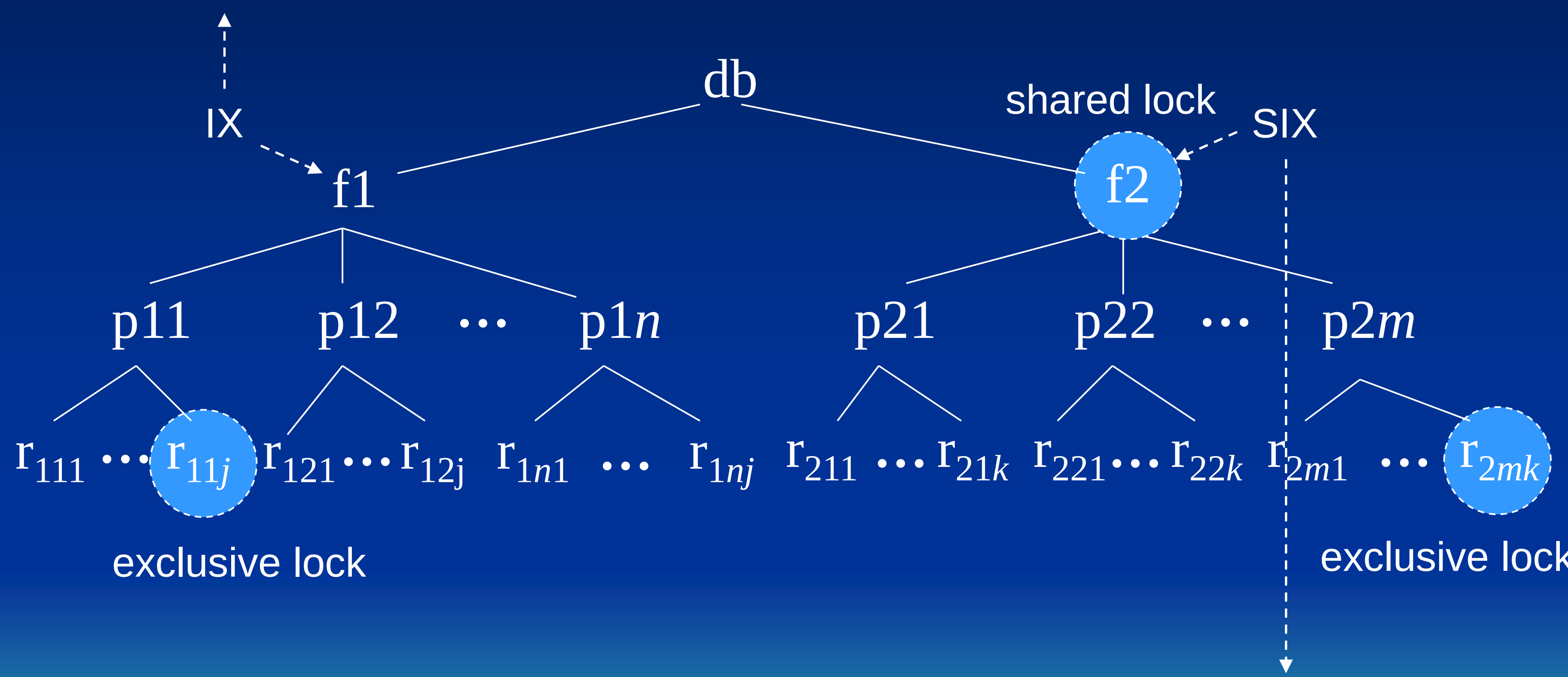
  Three types of intention locks:
  1. Intention-shared (IS) indicates that a shared lock(s) will be requested on some descendant node(s).
  2. Intention-exclusive (IX) indicates that an exclusive lock(s) will be requested on some descendant node(s).
  3. Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendant node(s).

If a transaction needs to lock some data item x, it will lock all those nodes on the path from the root of the granularity tree to the node x.

Indicate that a shared lock(s) will be requested on some descendant node(s).



Indicate that an exclusive lock(s) will be requested on some descendant node(s).



Indicate that the current node is locked in shared mode but an exclusive lock(s) will be requested on some descendant node(s).

Granularity of Data Items and Multiple Granularity Locking

- Lock compatibility matrix for multiple granularity locking

	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

If a transaction needs to lock some data item x, it will lock all those nodes on the path from the root of the granularity tree to the node x.

Granularity of Data Items and Multiple Granularity Locking

- Multiple granularity locking (MGL) protocol:

1. The lock compatibility must be adhere to.
2. The root of the granularity hierarchy 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 of 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.

Granularity of Data Items and Multiple Granularity Locking

- Example:

T1: updates all the records in file f1.

T2: read record r_{1nj}.

T1:
IX(db)
X(f1)
write-item(f1)
unlock(f1)
unlock(db)

T2:
IS(db)
IS(f1)
IS(p1*n*)
S(r_{1nj})
read-item(r_{1nj})
unlock(r_{1nj})
unlock(p1*n*)
unlock(f1)
unlock(db)

Granularity of Data Items and Multiple Granularity Locking

T2:	T1:
IS(db) IS(f1) IS(p1n) S(r_{1nj})	
	IX(db) X(f1)
read-item(r_{1nj}) unlock(r_{1nj}) unlock(p1n) unlock(f1) unlock(db)	
	write-item(f1) unlock(f1) unlock(db)

Concurrency - other topics

- Phantoms

  a phantom with respect to transaction T1 is a new record that comes into existence, created by a concurrent transaction T2, that satisfies a search condition used by T1.

- consider transactions that include the following operations:

<u>T1</u>

SELECT * FROM a
WHERE id BETWEEN 5 AND 10

<u>T2</u>

INSERT INTO a
VALUES (id, name) (7, ‘joe’)

a

Id	name
1	… …
2	… …
3	…
5	…
6	…
10	… …

insert (7, ‘joe’)

Concurrency - other topics

- Interactive transactions

values written to a user terminal prior to the commit point of a transaction T could be used as input to other transactions

this inter-transaction dependency is outside the scope of any DBMS concurrency controls

Concurrency - in SQL databases

- SQL isolation levels

If write lock is kept till T is committed, but read lock on any item X cannot be released until all read operations on X have been conducted.

SET TRANSACTION

< SERIALIZABLE |

REPEATABLE READ |

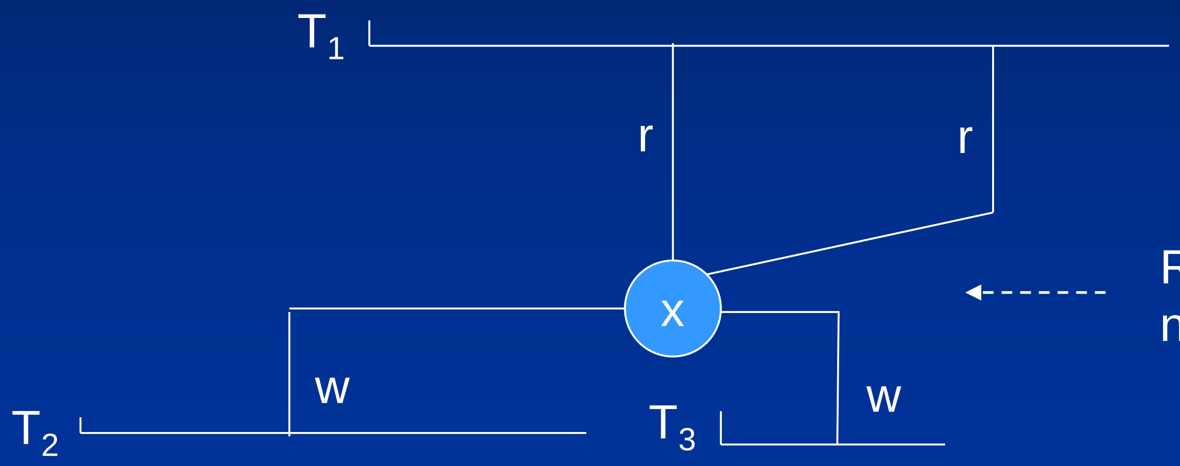
READ COMMITTED |

READ UNCOMMITTED >

If write lock is kept till T is committed, but read lock can be released earlier.

Reference: Data and databases: concepts in practice; Joe Celko; 1999; Morgan Kaufmann Publishers; ISBN 1-55860-432-4

Why is the “repeatable read” higher than the “read committed”?



Read committed data, but not repeatable read.

Concurrency - SQL

Phenomena	description
P1	dirty read (transaction can read data that is not committed)
P2	nonrepeatable read (transaction can read the same row twice, but it could be different)
P3	phantom

Concurrency - SQL

	Phenomena occurs?		
	P1	P2	P3
serializable	no	no	no
repeatable read	no	no	yes
read committed	no	yes	yes
read uncommitted	yes	yes	yes

- A Sample SQL Transaction

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
        READ WRITE
        ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
        VALUE (‘Robert’, ‘Smith’, ‘991004321’, 2, 75000);
EXEC SQL UPDATE EMPLOYEE
        SET SALARY = SALARY * 1.1 WHERE DNO = 2;
```

- A Sample SQL Transaction

```
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ...;
```