Concurrency Control

Lock-based Concurrency Control

- In general, a time-sharing OS/DMBS executes a set of programs in a random order
  - ready to run, suspended, running
- A concurrency control protocol must ensure that the sequence of read/write operations from transactions forms a serializable and recoverable schedule

Strict 2-Phase Lock (strict 2PL)

1. A transaction must obtain an S (shared) lock on an object before reading, and an X (exclusive) lock on an object before writing
2. If a transaction T holds an X lock on an object O, no other transactions can get a lock (S or X) on O
   a) If an object does not have a lock, a transaction can get a lock on it
   b) If an object has a shared lock, a transaction can get another shared lock, or an exclusive lock if the shared lock is owned by the transaction itself (lock upgrade)
   c) If an object has an exclusive lock, no transaction can get any lock
3. If a transaction's request is not granted, it is suspended
4. When a transaction finishes, all locks it holds are released
5. When a lock on O is released, resume the transactions waiting for a lock on O

Example 1

P1	P2
R(A)	R(A)
W(A)	commit/abort
commit/abort	

Lock insertions

1) One lock per object: DBMS needs to know all read/write operations in advance (e.g., which can be done at the compiling time, etc.)

2) One lock per read/write: Just need to insert a lock inside the APIs for read/write

P1	P2
X(A)	S(A)
R(A)	R(A)
W(A)	commit/abort
commit/abort	

P1	P2
S(A)	S(A)
R(A)	R(A)
X(A)	commit/abort
W(A)	
commit/abort	

Example 1

P1	P2
S(A)	S(A)
R(A)	R(A)
X(A)	commit/abort
W(A)	
commit/abort	

T1	T2
	S(A)
	R(A)
S(A)	
R(A)	
X(A)	
	c/a
W(A)	
c/a	

Lock table

Data	*Lock*	*Owner*	*Waiting*
A	X	T_1	

Example 1

P1	P2
S(A)	S(A)
R(A)	R(A)
X(A)	commit/abort
W(A)	
commit/abort	

T1	T2
S(A)	
R(A)	
	S(A) ?
	R(A)
X(A)	
	c/a
W(A)	
c/a	

Lock table

Data	*Lock*	*Owner*	*Waiting*
A	X	T_1	

Example 1

P1	P2
S(A)	S(A)
R(A)	R(A)
X(A)	commit/abort
W(A)	
commit/abort	

T1	T2
S(A)	
R(A)	
X(A)	
W(A)	
c/a	
	S(A)
	R(A)
	c/a

T1	T2
	S(A)
	R(A)
	c/a
S(A)	
R(A)	
X(A)	
W(A)	
c/a	

T1	T2
	S(A)
	R(A)
S(A)	
R(A)	
	c/a
X(A)	
W(A)	
c/a	

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
	c/a
X(A)	
W(A)	
c/a	

Example 2

P1	P2
R(A)	R(A)
R(C)	R(B)
W(C)	W(B)
c/o	c/o

P1	P2
S(A)	S(A)
R(A)	R(A)
S(C)	S(B)
R(C)	R(B)
X(C)	X(B)
W(C)	W(B)
c/o	c/o

T1	T2

S(A)
R(A)	

	S(A)
	R(A)

S(C)
R(C)	

	S(B)
	R(B)

X(C)
W(C)	

	X(B)
	W(B)

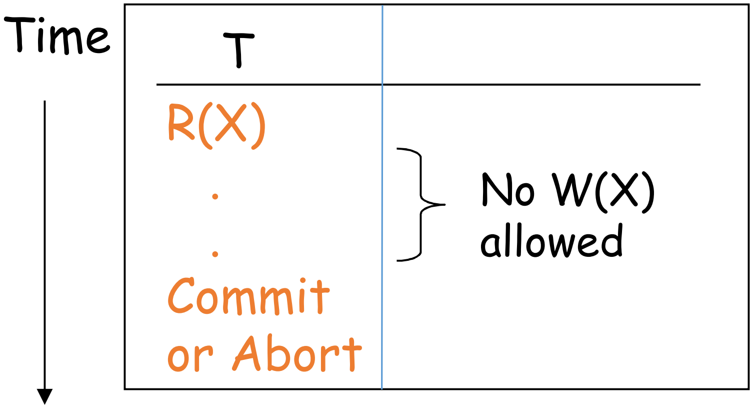
Data	*Lock*	*Owner*	*Waititing*
A	S	T_1 , T_2	
C	X	T_1	
B	X	T_2	

c/a

c/a

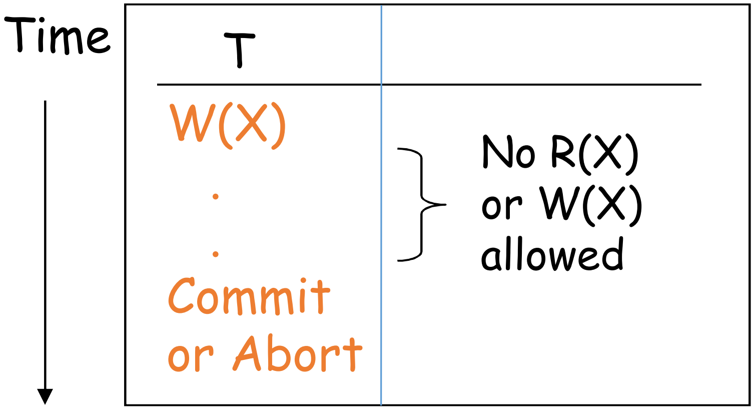
The effect of Strict 2PL

Strict Schedule



Time

T	
R(X) . . Commit or Abort	No W(X) allowed

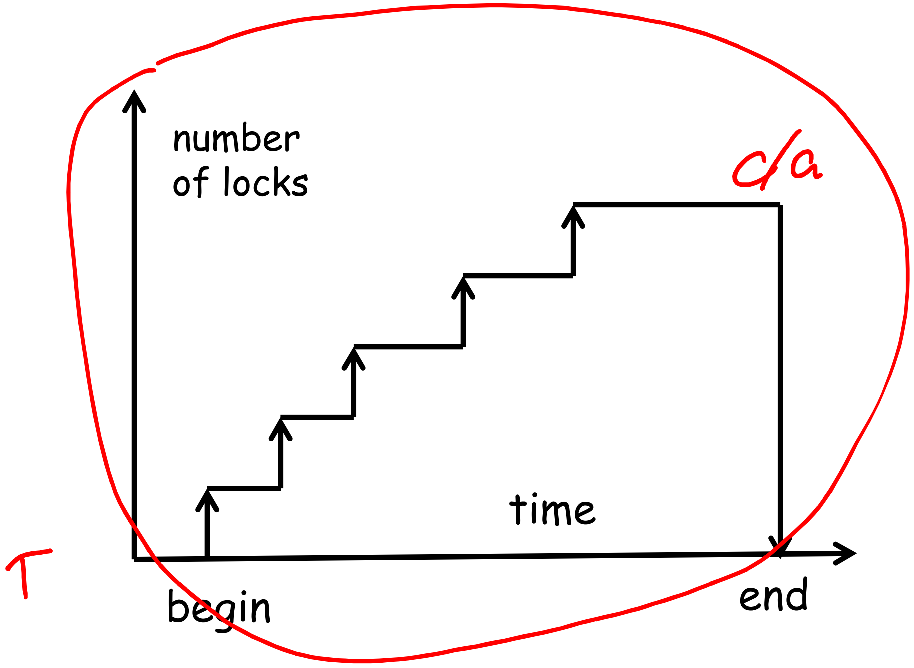


Time

T	
W(X) . . Commit or Abort	No R(X) or W(X) allowed

Strict 2PL ensures that the order of read/write operations from a set of transactions forms a strict schedule

1. It avoids RW, WR, WW conflicts, and
2. It does not require cascading aborts, and actions of an aborted transaction can be undone.



Deadlocks

- Transactions may waiting for each other and create deadlocks

T1	T2
S(A)	
R(A)	
	S(A)
	R(A)
X(A)	
	X(A)
abort	
	~~W~~ W(A)
	commit

Example 3

P1	P2

R(A)
W(A)
commit/abort
	R(A)
	W(A)
	commit/abort

P1	P2

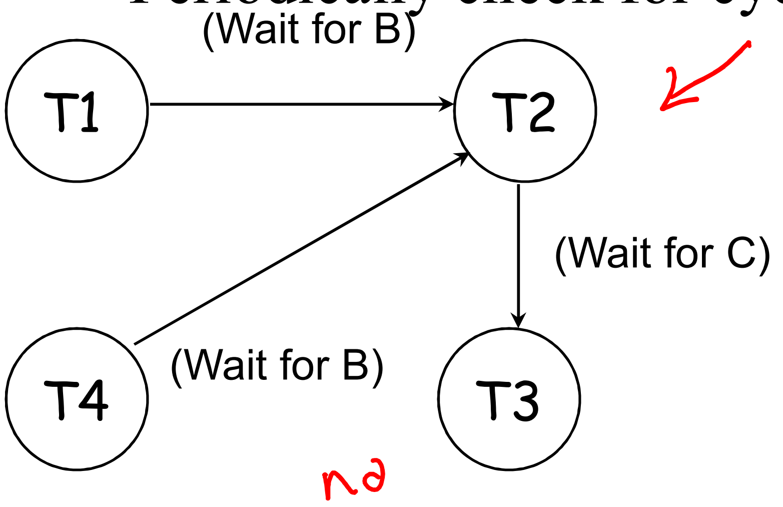
S(A)
R(A)
X(A)
W(A)
commit/abort
	S(A)
	R(A)
	X(A)
	W(A)
	commit/abort

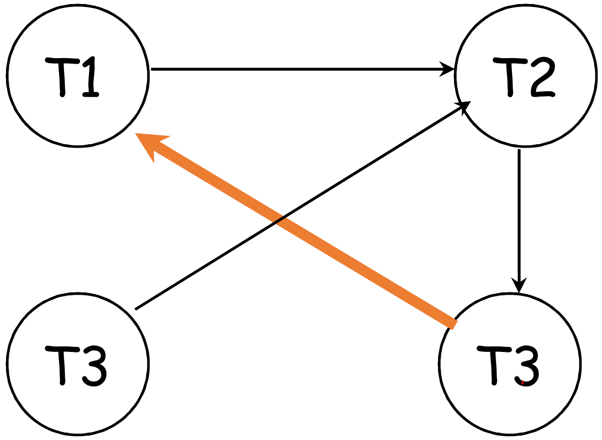
S(A)
R(A)

Data	*Lock*	*Owner*	*Waiting*
A			

Dealing with Deadlock 1: Detection

- Use a timeout mechanism
  - If a transaction has been waiting for too long, abort the transaction
- Transaction manager maintains a wait-for graph:
  - Nodes correspond to active transactions
  - Add an edge from Ti to Tj if Ti is waiting for Tj to release a lock
  - Remove an edge when a lock request is granted
  - Periodically check for cycles in the waits-for graph





T1	T2	T3	T4

S(A)
R(A)			

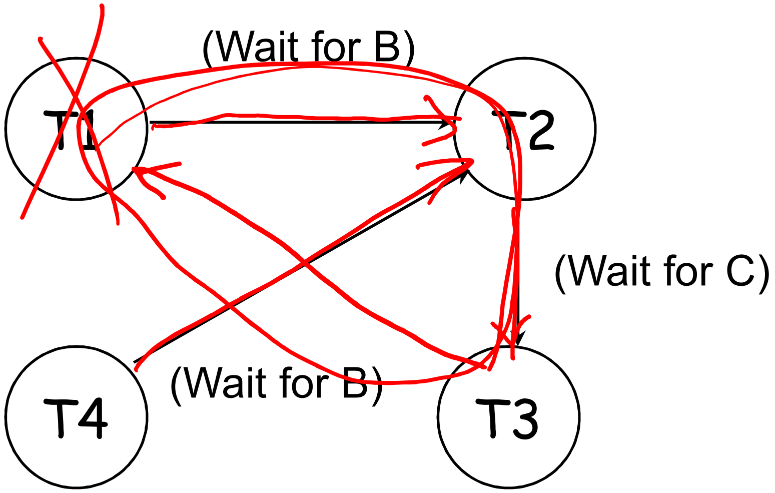
	X(B)
	W(B)		

~~S(B)~~			

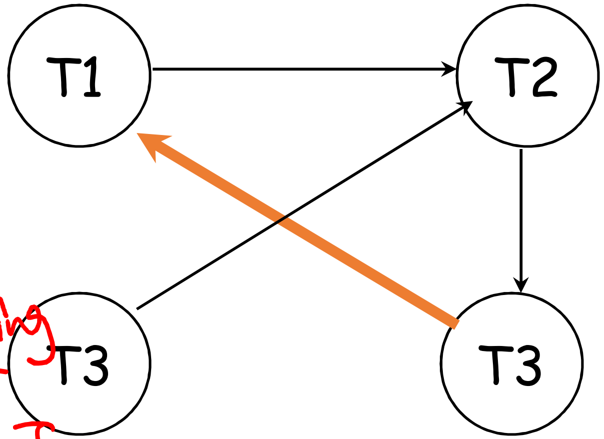
		S(C)
		R(C)	

	X(C)		
			X(B)
		X(A)	

Wait-for graph



Cyclic Deadlock →



Lock table

data	Lock	owner	waiting
A	S	T_1	T_3
B	X	T_2	T_1, T_4
C	S	T_3	T_2

Dealing with Deadlock 2: Prevention

- Assign priorities based on timestamps
  - The lower the timestamp, the higher is transaction's priority
- Assume Ti wants a lock that Tj holds
  - Wait-die: (the older waits for the younger)
    - If the older has higher priority, let it wait;
    - Otherwise, abort the older
  - Wound-wait: (the younger waits for the older)
    - If the older has higher priority, abort the younger;
    - Otherwise, let it wait
- If a transaction re-starts (younger transaction restarts), make sure it has its original timestamp so that no transaction is perennially aborted

Performance of Locking

- Locked-based schemes resolve conflict using blocking and aborting, both incurring performance penalty
    - Blocked transactions may hold locks that force other transactions to wait
    - Aborted transactions need to be rolled back and restarted
- Increasing the number of transactions will initially increase the concurrency, but when the number of deadlocks increase to certain level (i.e., thrashing), the performance starts to downgrade

Relevant Questions with Lock-Based Concurrency Control

- Should we use deadlock prevention or deadlock detection?
- How frequently should we check for deadlocks?
- When deadlock occurs, which transaction should be aborted?
- Detection-based schemes work well in practice.
- Choice of deadlock victim to be aborted:
  - Transaction with fewest locks
  - Transaction that has done the least work
  - Transaction that is farthest from completion
  - There is a rich literature on this topic

Example 1

P1	P2
R(A)	R(A)
W(A)	commit/abort
commit/abort	

Lock insertions

1) One lock per object: DBMS needs to know all read/write operations in advance (e.g., which can be done at the compiling time, etc.)

2) One lock per read/write: Just need to insert a lock inside the APIs for read/write

P1	P2
X(A)	S(A)
R(A)	R(A)
W(A)	commit/abort
commit/abort	

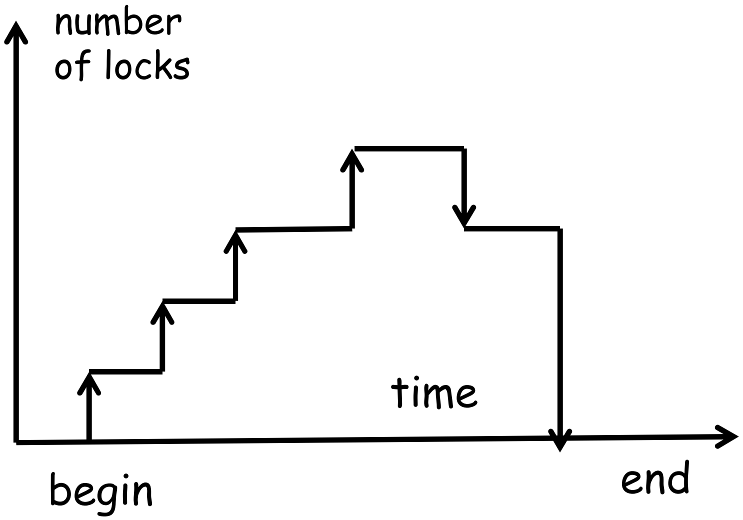
X(A) ✓

W(A)

P1	P2
S(A)	S(A)
R(A)	R(A)
X(A)	commit/abort
W(A)	
commit/abort	

Two-Phase Locking (2PL)

1. Each transaction must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
2. A transaction can not request additional locks once it releases any locks.



2PL allows more concurrency, but is difficult to implement

- Necessary locks may be identified during the compiling phase
- But during the run time, need to know when the transaction has obtained all its locks
- Some schedules may be unrecoverable, a major problem

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
Abort	

Using Strict 2PL, the following schedule is not allowed.

T1	T2
X(A)	
R(A)	
W(A)	
←	X(A)
c/a	

Using 2PL, the following unrecoverable schedule is allowed.

T1	T2
X(A)	
R(A)	
W(A)	
	X(A)
	R(A)
	X(B)
	R(B)
	W(B)
	Commit
Abort	

X(A) is released.

X(A) and X(B) are released.

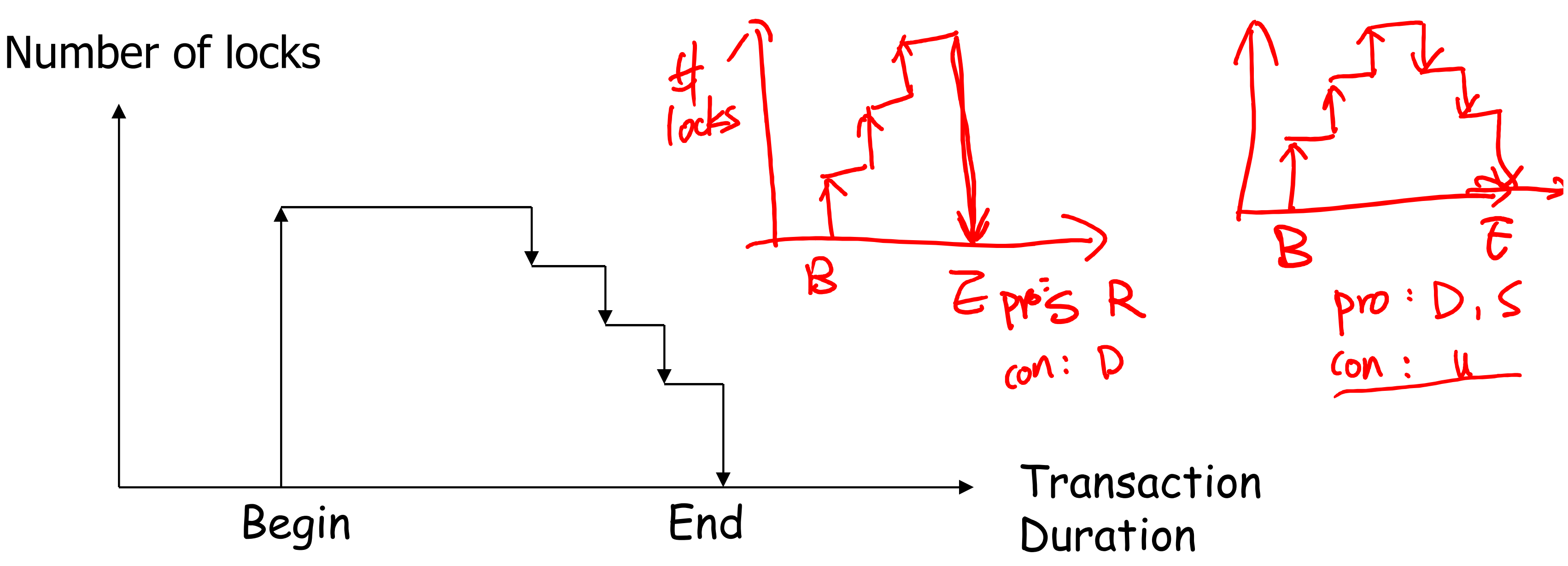
Conservative 2PL

Technique: A transaction obtains all the locks that it will need when it begins. If it does not get all the locks, it does not hold any lock.

Advantage: Can also prevent deadlocks.

No rollback

Disadvantage: May not recoverable

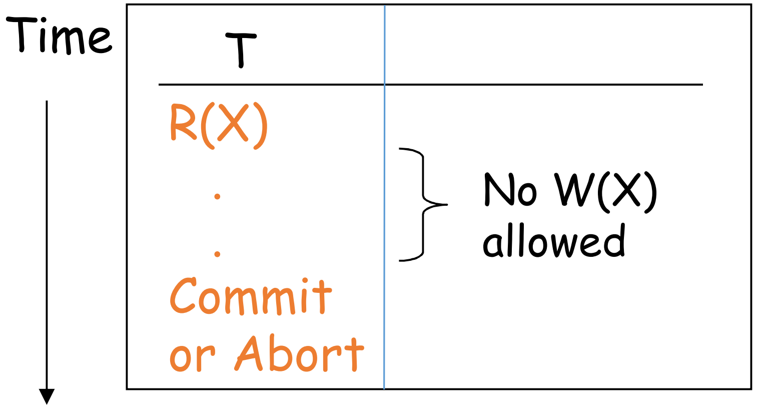


Concurrency Control without Locking

- Timestamp-Based Concurrency Control
- Optimistic Concurrency Control
  - Assumption: Most transactions will not conflict with other transactions
  - Not widely used
- Multiversion Concurrency Control
  - Give serializable schedules
  - Used in Oracle 8

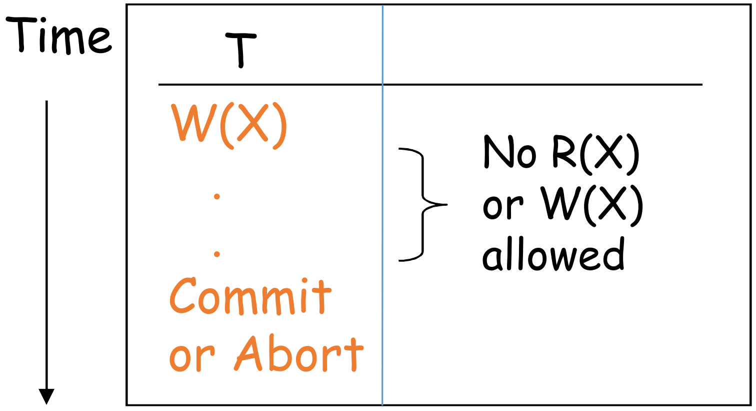
The effect of Strict 2PL

Strict Schedule



Time

T	
R(X) . . Commit or Abort	No W(X) allowed

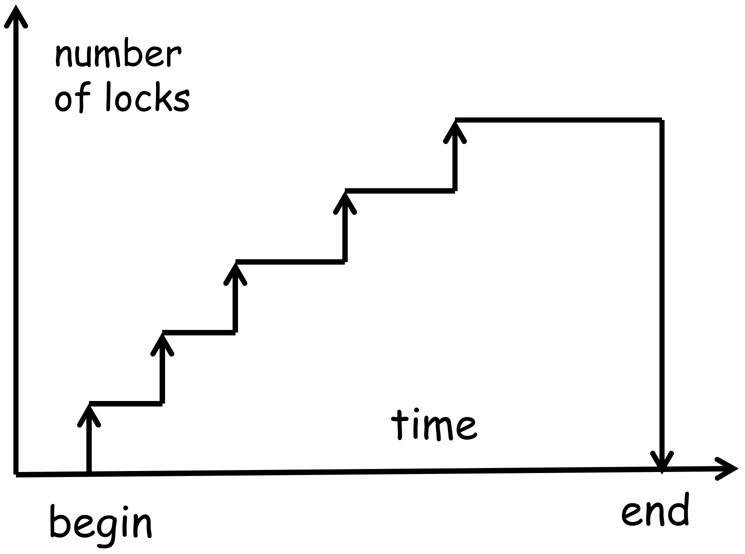


Time

T	
W(X) . . Commit or Abort	No R(X) or W(X) allowed

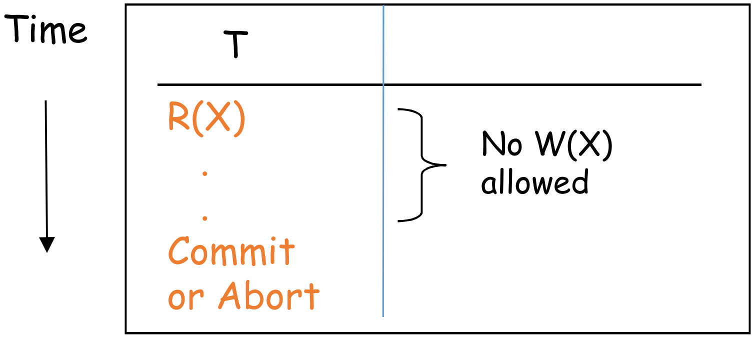
Strict 2PL ensures that the order of read/write operations from a set of transactions forms a strict schedule

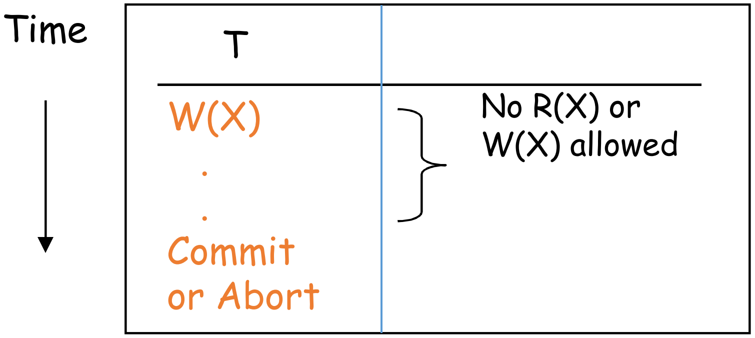
1. It avoids RW, WR, WW conflicts, and
2. It does not require cascading aborts, and actions of an aborted transaction can be undone.



Quick Review

- Strict schedule

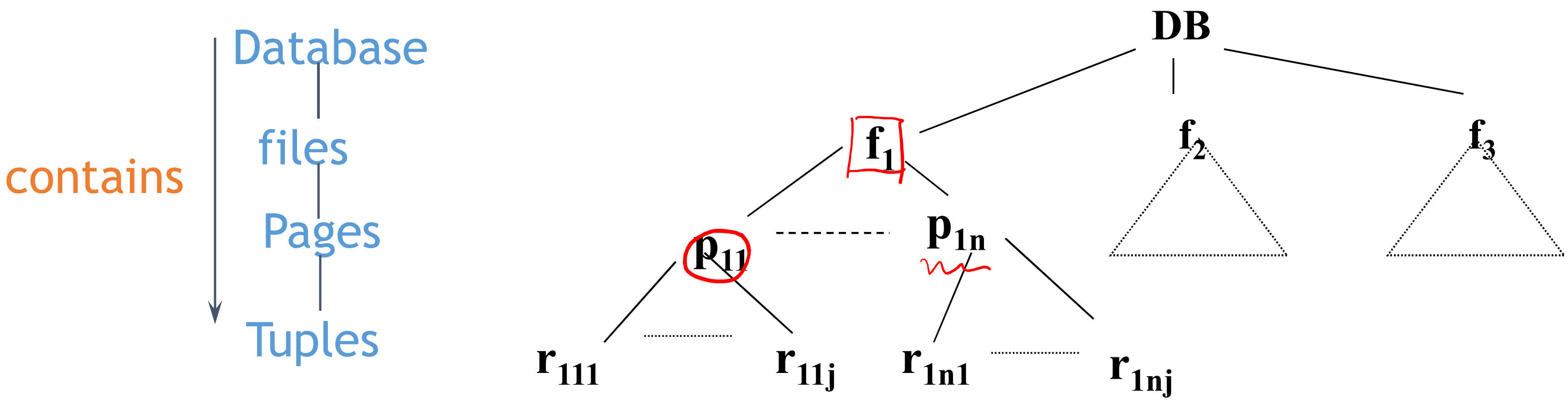




- Strict 2-phase locking
  1. A transaction must obtain an S lock on an object before reading, and an X lock on an object before writing
  2. If a transaction T holds an X lock on an object, no other transactions can get a lock on the object
  3. If a transaction's request is not granted, it is suspended
  4. When a transaction finishes, all locks it holds are released
  5. When a lock on an object is released, resume the transactions waiting for a lock on the object

Strict 2PL ensures that the execution order of the read/write operations from a set of transaction forms a strict schedule

Multiple-Granularity Locking (MGL)



contains

Database

files

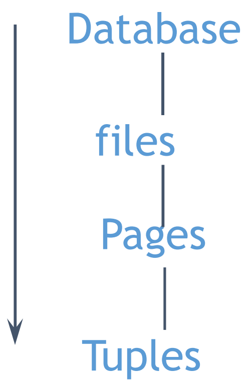
Pages

Tuples

- Which granularity should the DBMS provide concurrency control?
  - Coarse Granularity means less concurrency
  - Fine Granularity incurs more lock management overhead
- How ensure that an object is not locked by conflicting locks at a different granularity?

Motivation Example

Containment hierarchy

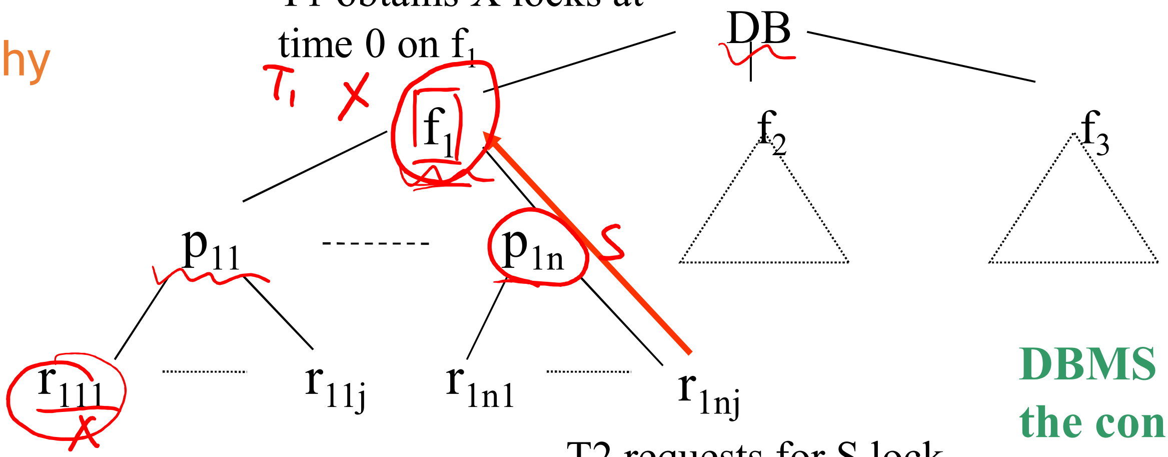


Database

files

Pages

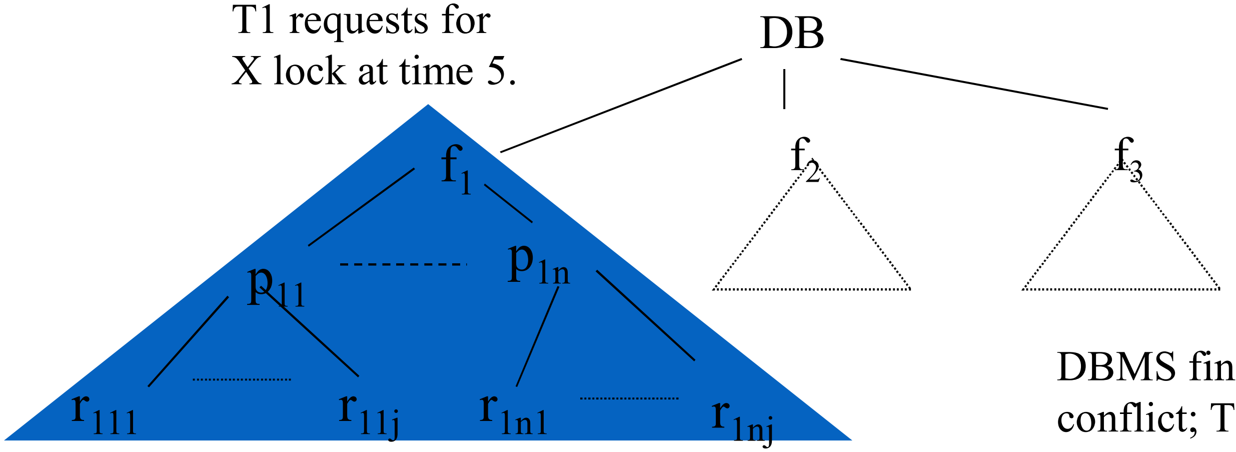
Tuples



T1 obtains X locks at time 0 on f_1

T2 requests for S lock at time 5.

DBMS can find the conflict efficiently and block T2.



T1 requests for X lock at time 5.

T2 obtains an S lock at time 0.

DBMS finds the conflict; T2 must wait.

DBMS must traverse the subtree of f_1 to check for conflicting locks.

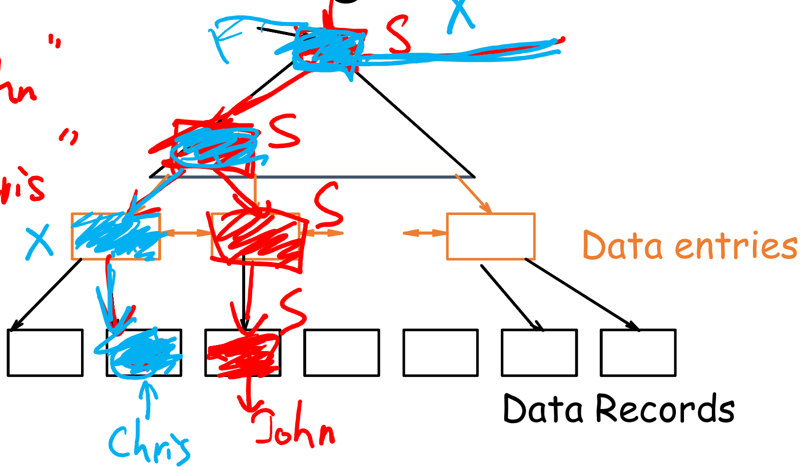
A Special Case: B+ Tree

- How can we efficiently lock a particular leaf node?
- One solution: Ignore the tree structure, just lock pages while traversing the tree, following 2PL.
- This has terrible performance!
  - Root node (and many higher level nodes) become bottlenecks because every tree access begins at the root.

select name = "John"

T_2 update ~~select~~ name = "Chris"

READ

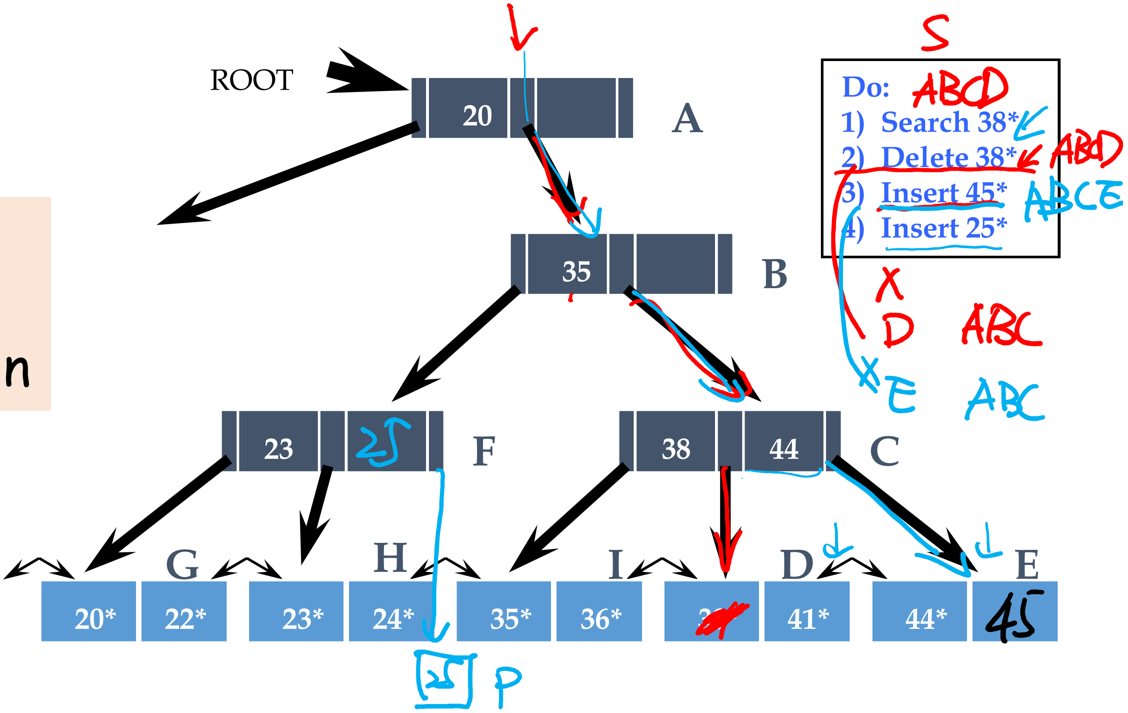


Two Useful Observations

- Higher levels of the tree only direct searches for leaf pages
- When inserting a data item, a node on the path from root to the leaf node won't be changed unless its child node is full
- When deleting a data item, a node on the path from root to the leaf node won't be changed unless its child node is half-empty

B+ tree

No need to lock a node unless it may be impacted by an insertion or a deletion

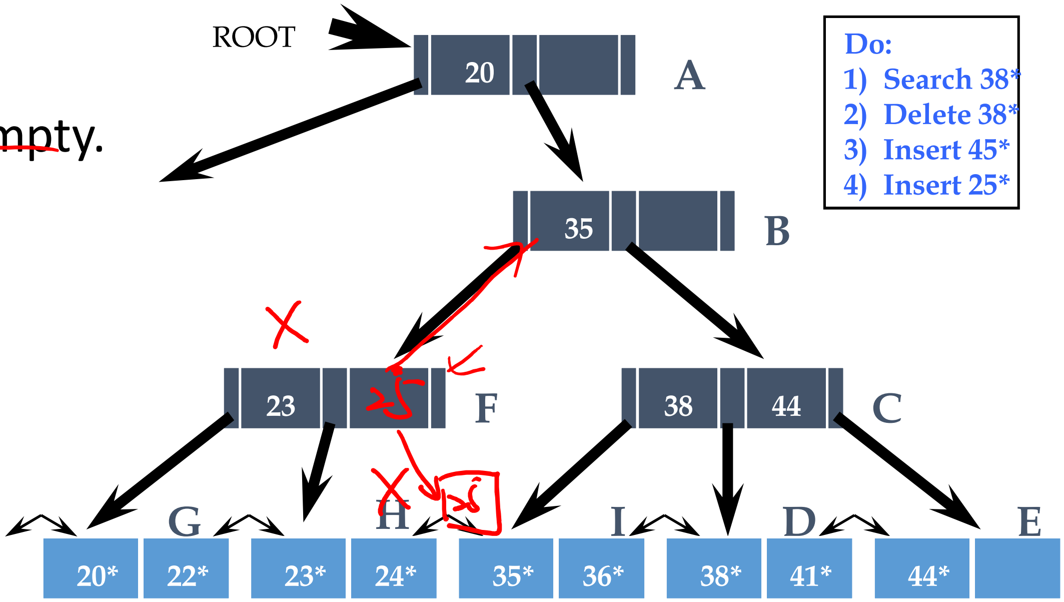


Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*

A Simple Tree Locking Algorithm

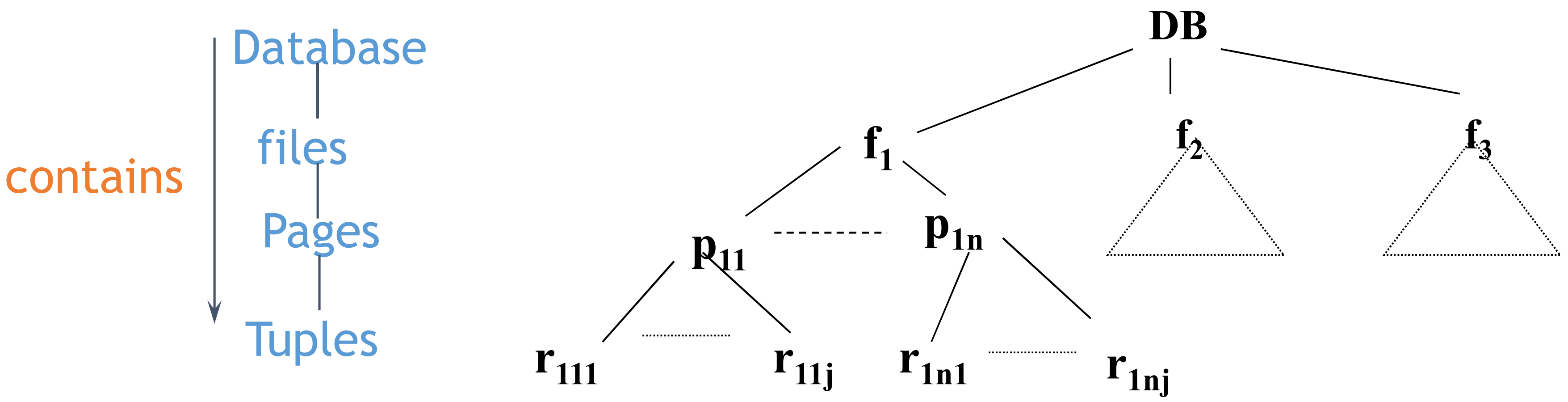
2PL

- Search: Start at root and go down; repeatedly, S lock child then unlock parent.
- Insert/Delete: Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:
  - If child is safe, release all locks on ancestors.
- Safe node: Node such that changes will not propagate up beyond this node.
  - Inserts: Node is not full.
  - Deletes: Node is not half-empty.



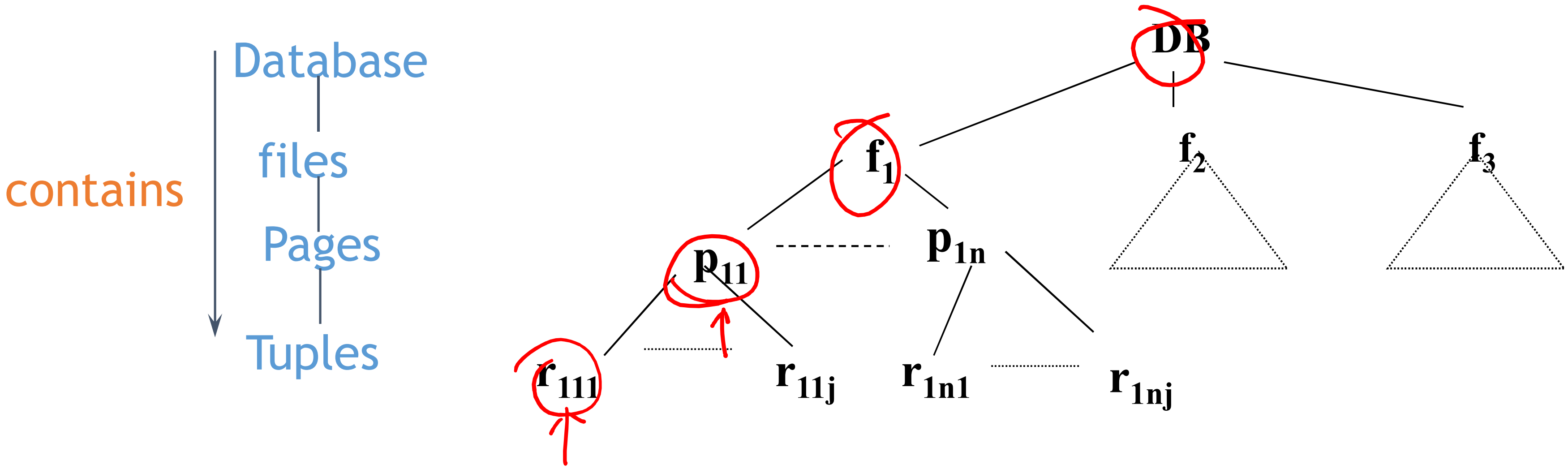
Do:
1) Search 38*
2) Delete 38*
3) Insert 45*
4) Insert 25*

General Case



- Same as before, using two locks, shared and exclusive
  - Before reading, have a share lock on all nodes along the path
  - Before writing, have an exclusive lock on all nodes along the path
- Performance suffers

Solution: Adding new locks



contains

Database

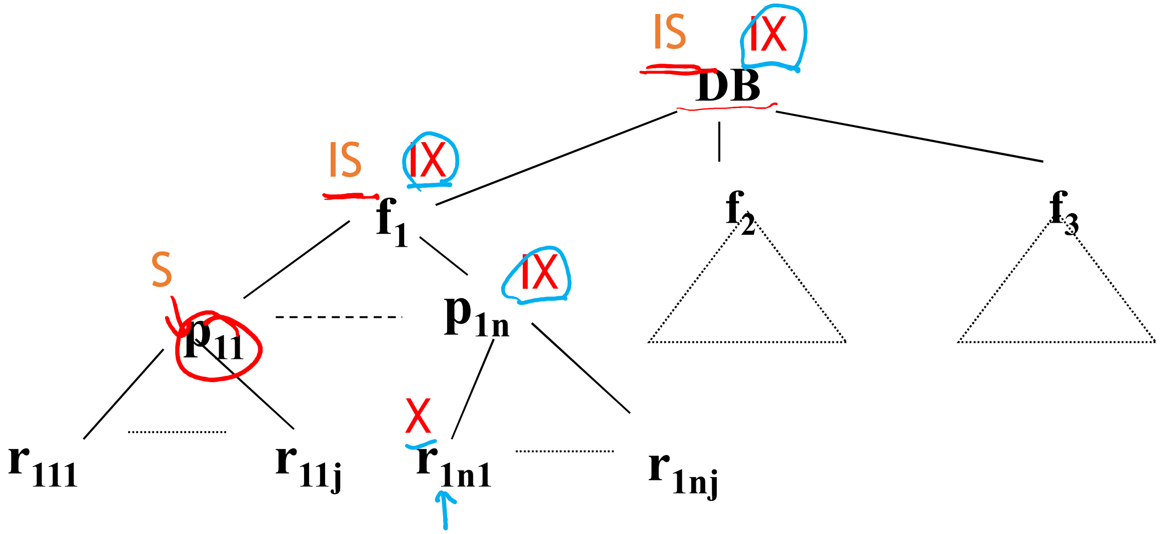
files

Pages

Tuples

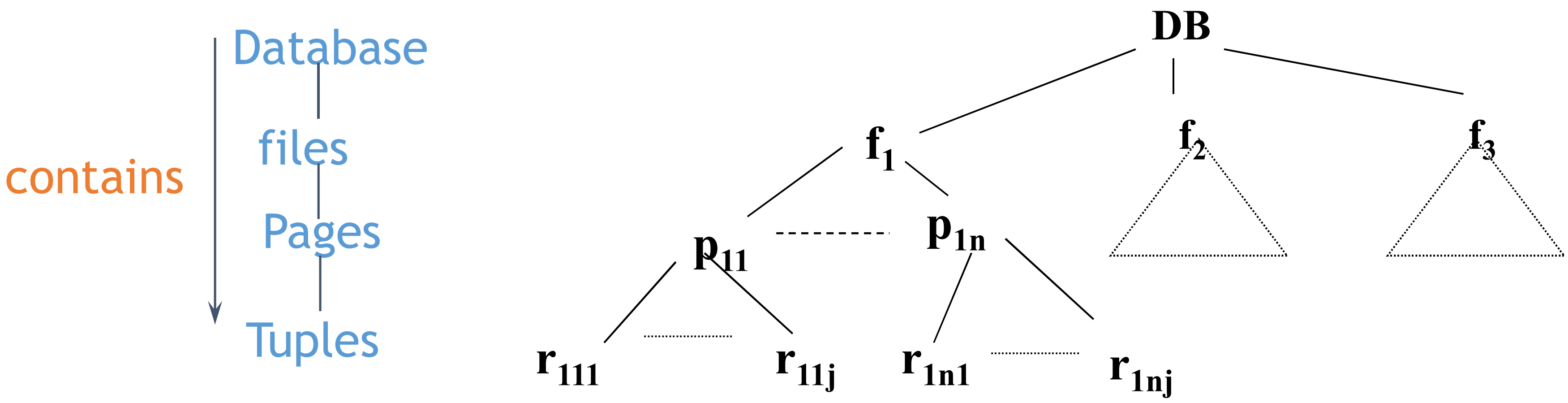
- IS: intention shared (IS) lock
  - Indicate that a transaction will apply for a share (S) lock underneath
- IX: intention exclusive (IX) lock
  - Indicate that a transaction will apply for an exclusive (X) lock underneath

Locking Protocol



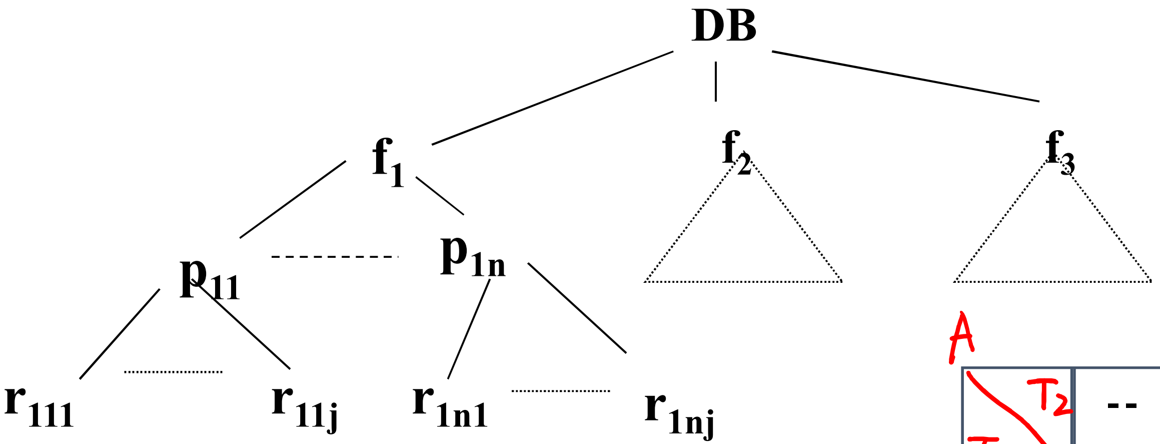
1. Before share lock an object, have an IS lock on each node along the path from the root to the object
2. Before exclusive lock an object, have an IX lock on each node along the path from the root to the object

Locking Protocol



1. Locking starts from the root node
2. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode
3. A node N can be locked by a transaction T in X or IX mode only if the parent node of N is already locked by transaction T in IX mode
4. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
5. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T. (i.e., unlocking starts from bottom up).

Lock Compatibility

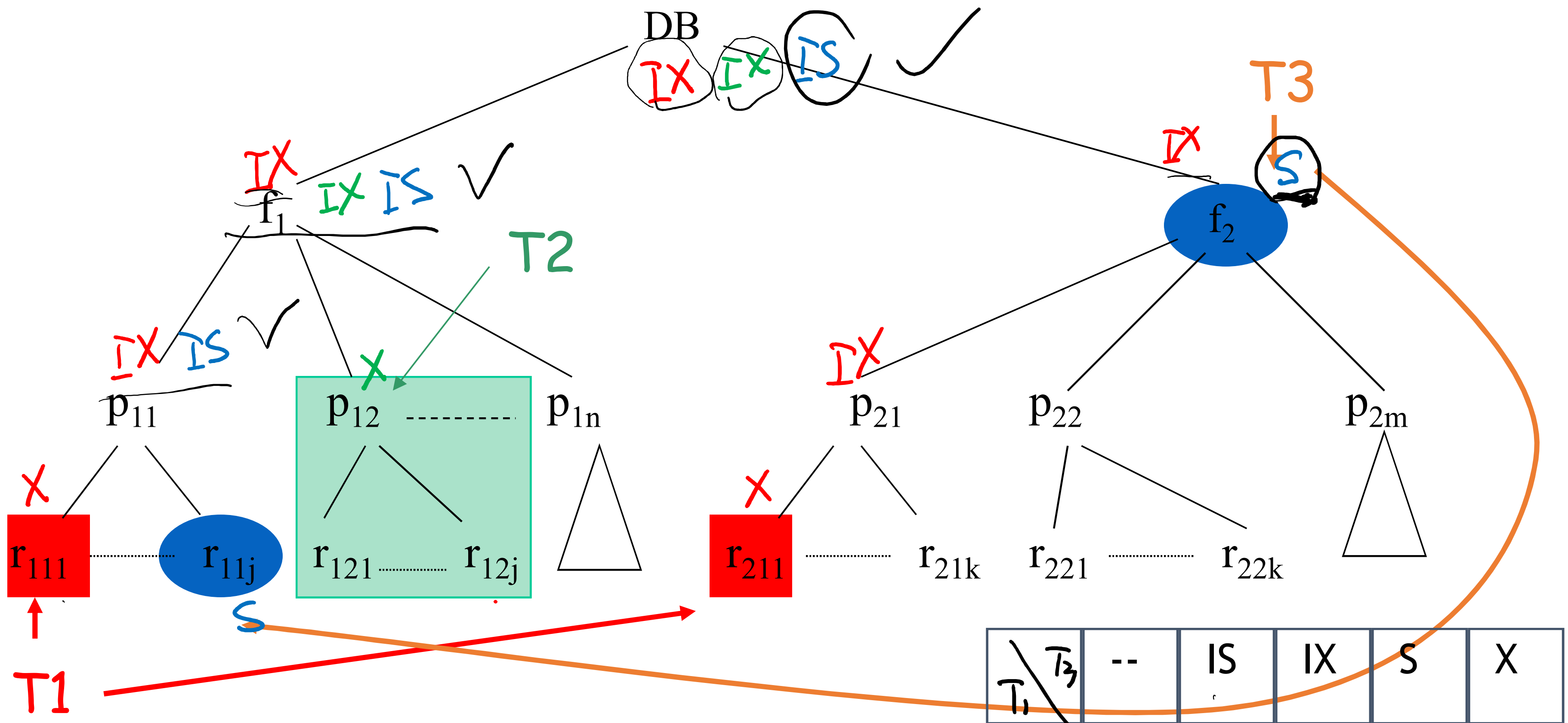


A

T_2 T_1	--	IS	IX	S	X
--	OK	OK	OK	OK	OK
IS	OK	OK	OK	OK	
IX	OK	OK	OK		
S	OK	OK		OK	
X	OK				

DBMS

IS and IX are compatible!



T1 \ T3	--	IS	IX	S	X
--	OK	OK	OK	OK	OK
IS	OK	OK	OK	OK	
IX	OK	OK	OK		
S	OK	OK		OK	
X	OK				

Three transactions

1st T1 updates r_{111} and r_{211} RED

2nd T2 updates all records in P_{12} GREEN

3rd T3 reads r_{11j} and the entire f_2 BLUE

T1	T2	T3
IX(db)	IX(db)	IS(db)
IX(f_1)	IX(f_1)	IS(f_1)
IX(p_{11})	X(p_{12})	IS(p_{11})
X(r_{111})	W(r_{121})	S(r_{11j})
W(r_{111})	…	R(r_{11j})
IX(f_2)	W(r_{12j})	S(f_2)
IX(p_{21})	Unlock(p_{12})	R(f_2)
X(r_{211})	Unlock(f_1)	Unlock(r_{11j})
W(r_{211})	Unlock(db)	Unlock(f_1)
Unlock(r_{111})		Unlock(f_2)
Unlock(p_{11})		Unlock(db)
Unlock(f_1)	…	
Unlock(r_{211})		
Unlock(p_{21})		
Unlock(f_2)		
Unlock(db)		

A Serializable Schedule

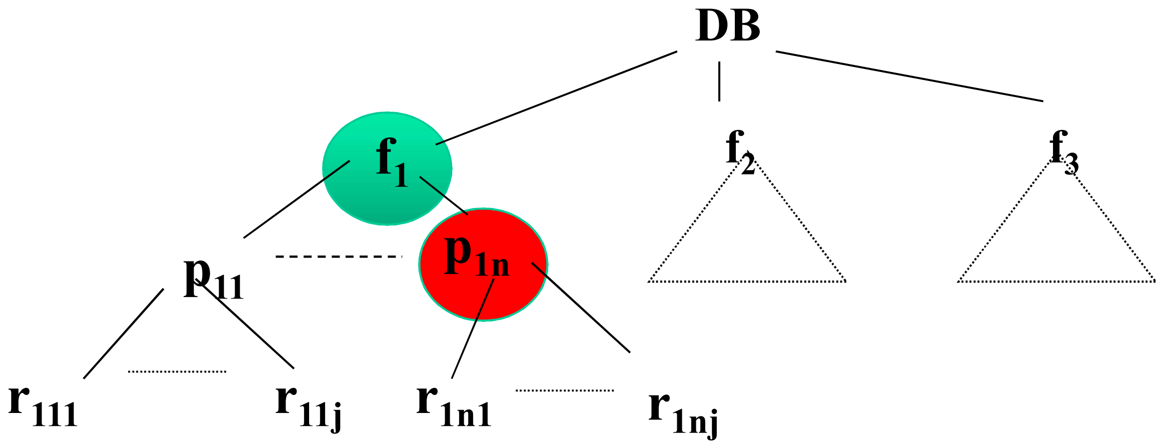
🟦 Does not block each other

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)		
🟥 X(r211)		
Unlock(r211)		
Unlock(p21)		
Unlock(f2)		
		🟥 S(f2)
	Unlock(p12)	
	Unlock(f1)	
	Unlock(db)	
Unlock(r111)		
Unlock(p11)		
Unlock(f1)		
Unlock(db)		
		Unlock(r11j)
		Unlock(p11)
		Unlock(f1)
		Unlock(f2)
		Unlock(db)

Improvement

It is quite common that a transaction needs to read an entire file and modify a few of the records in it.

In this case, it needs an S lock on the file and also an IX lock on the file (so that it can subsequently lock some of the contained objects in X mode)



- One more lock: Shared-Intention-eXclusive (SIX)
  - indicates that the current node is locked in a shared mode, but an exclusive lock(s) will be requested on some descendant node(s).
  - Logically, an SIX lock is equivalent to holding an S lock and an IX lock (but this is processed in one time, thus reducing lock cost)

Multiple-Granularity Locking (MGL)

- Intention-shared (IS) indicates that a shared lock(s) will be requested on some descendant node(s).
- Intention-exclusive (IX) indicates that an exclusive lock (s) will be requested on some descendant node(s).
- Shared-Intention-exclusive (SIX) indicates that the current node is locked in a shared mode, but an exclusive lock(s) will be requested on some descendant node(s).

	--	IS	IX	S	X	SIX
--	OK	OK	OK	OK	OK	OK
IS	OK	OK	OK	OK		OK
IX	OK	OK	OK			
S	OK	OK		OK		
X	OK					
SIX	OK	OK				

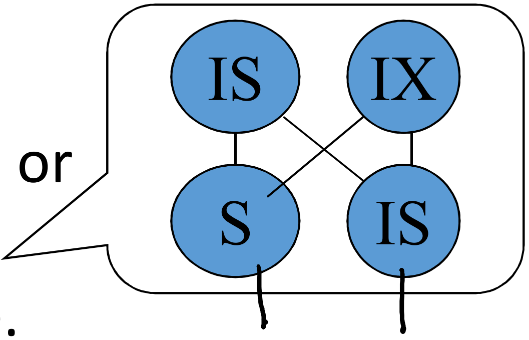
Lock compatibility matrix

MGL is good for a mixed type of transactions

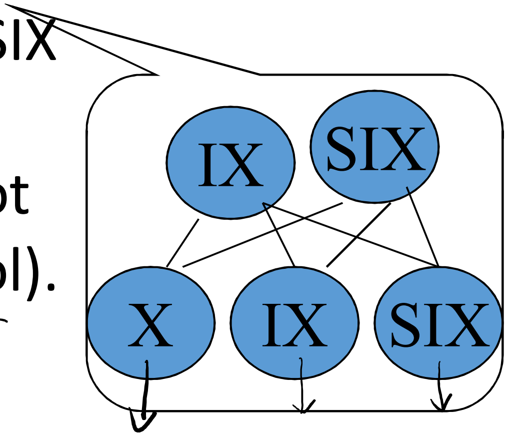
- Short transaction accessing few objects.
- Long transaction accessing the entire file.

Locking Protocol

1. Locking starts from the root node.
2. A node N can be locked by a transaction T in S or IS mode only if the parent node N is already locked by transaction T in either IS or IX mode.



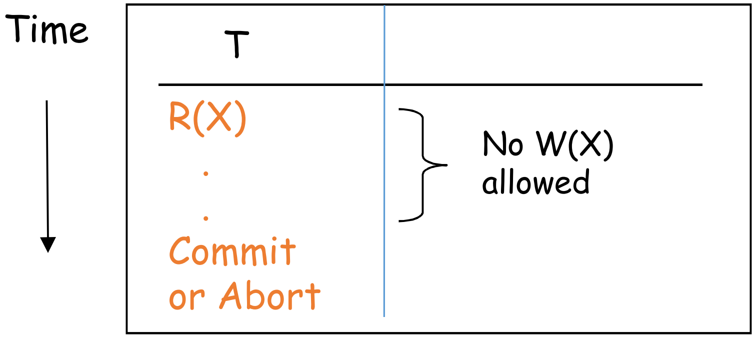
3. A node N can be locked by a transaction T in X, IX, or SIX mode only if the parent node of N is already locked by transaction T in either IX or SIX mode.



4. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).
5. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T. (i.e., unlocking starts from bottom up).

Quick Review

- Schedule, serial schedule, serializable and recoverable schedules
- Strict schedule



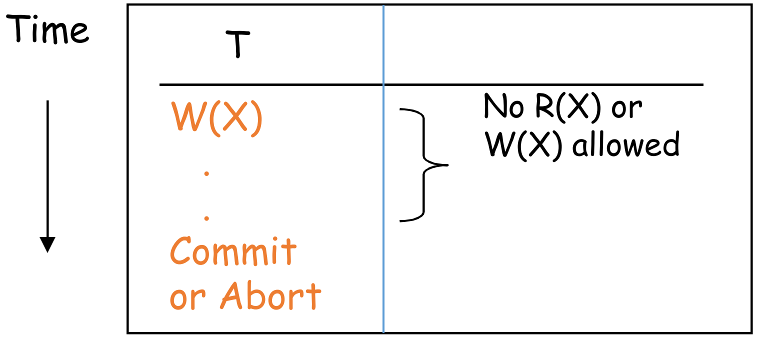
Time

T

R(X)

Commit or Abort

No W(X) allowed



Time

T

W(X)

Commit or Abort

No R(X) or W(X) allowed

- Strict 2-phase locking (Strict 2PL)
  1. A transaction must obtain an S lock on an object before reading, and an X lock on an object before writing
  2. If a transaction T holds an X lock on an object, no other transactions can get a lock on the object
  3. If a transaction's request is not granted, it is suspended
  4. When a transaction finishes, all locks it holds are released
  5. When a lock on an object is released, resume the transactions waiting for a lock on the object
- Multi-granularity locking (MGL)
  - B+-tree locking (Crabbing/Lock coupling)
  - General case (three new locks: IS, IX, SIX)