Lock-based Concurrency Control

SEI.ECNU

華東師範大學
EAST CHINA NORMAL UNIVERSITY

Concurrency Control

- A database must provide a mechanism that will allow txns to access a database in **a multi-programmed fashion** while preserving the illusion that each of them is executing alone on a dedicated system.
  - serializable, recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates **serial schedules**, but provides a **poor** degree of concurrency. **Why?**
- Goal – to develop concurrency control **protocols** that will assure serializability.

Concurrency Control Schemes

- Two-Phase Locking (**2PL**)
  - Assume txns will **conflict** so they must acquire locks on database objects **before** they are allowed to access them.
- Timestamp Ordering (**T/O**)
  - Assume that conflicts are **rare** so txns do not need to first acquire locks on database objects and instead **check** for conflicts at commit time.

Lock-Based Protocols

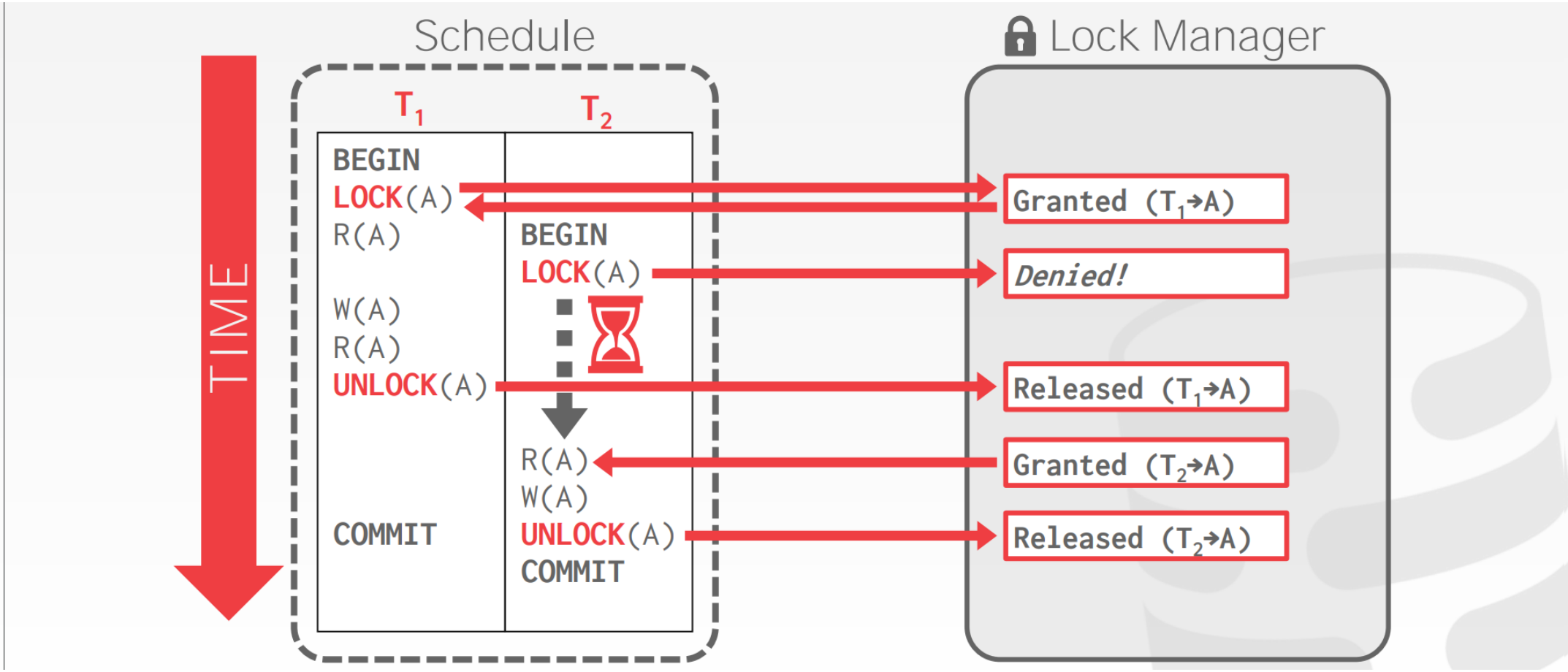
- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :
  1. **exclusive** *(X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. **shared** *(S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

	S	X
S	true	false
X	false	false

Lock-Compatibility Matrix

- Lock requests are made to **concurrency-control manager**.
  - Transaction can proceed only after request is granted.

Executing with Locks



Deadlock

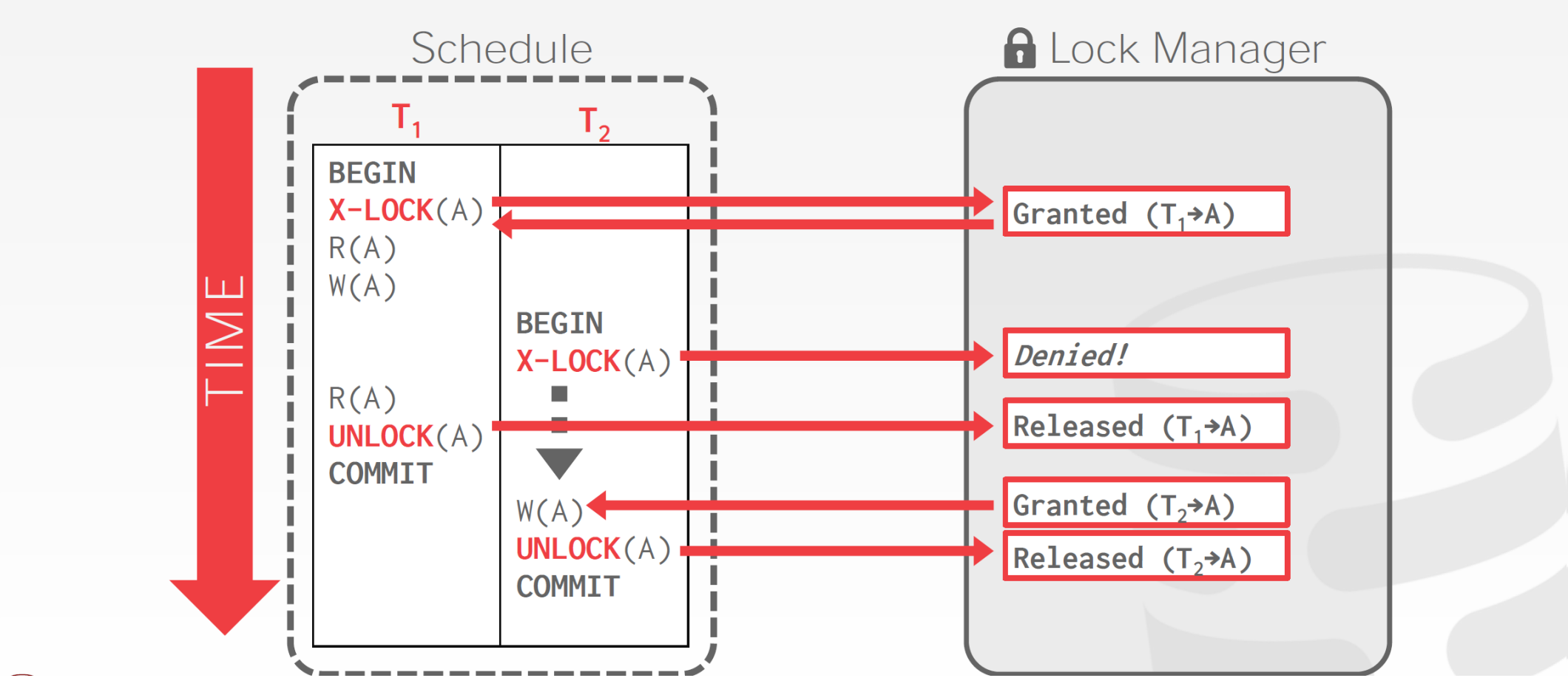
- Neither T_3 nor T_4 can make progress
  - Executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*
  - *E*xecuting **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**
  - One of T_3 or T_4 must be rolled back and its locks released.
- **Starvation** is also possible if concurrency control manager is badly designed

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Two-Phase Locking

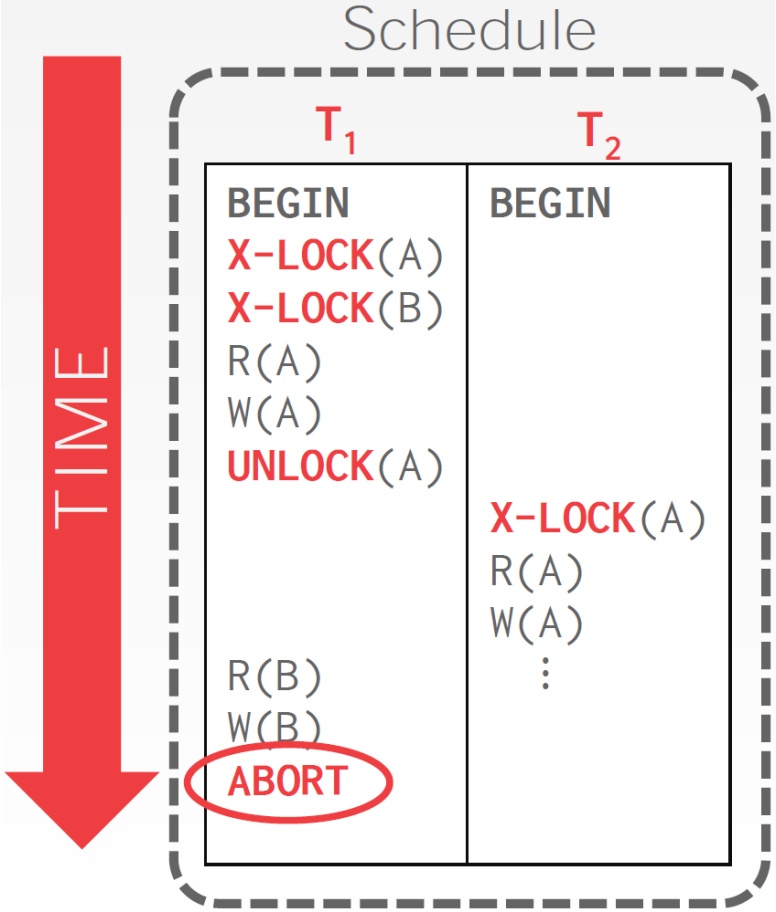
- A protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks
- 2PL is sufficient to guarantee **conflict serializability**.
  - It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

Executing with 2PL

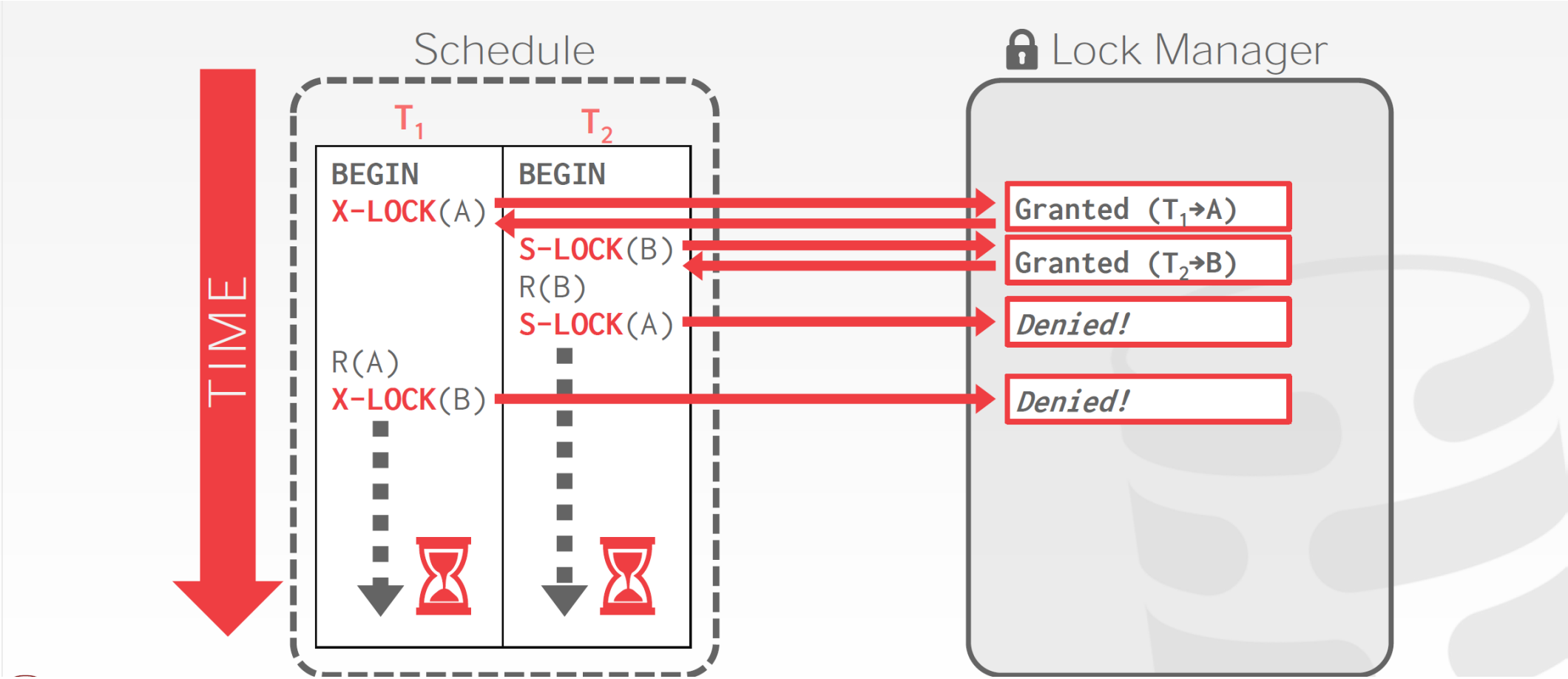


2PL – Cascading Aborts

- This is a **permissible** schedule in 2PL, but the DBMS has to also **abort** T_2 when T_1 aborts.
  - Any information about T_1 cannot be "leaked" to the outside world.



2PL – Deadlocks



Extensions to 2PL

- 2PL does not ensure freedom from cascading roll-back and deadlocks
- Extensions to basic 2PL
  - **Strict 2PL**: a transaction must hold all its exclusive locks till it commits/aborts.
    - Ensures recoverability and avoids cascading roll-backs
  - **Rigorous (Strong Strict) 2PL**: a transaction must hold all locks till commit/abort.
    - Transactions can be serialized in the order in which they commit.
- Most databases implement **rigorous 2PL**, but refer to it as 2PL

Lock Conversions

- Two-phase locking protocol with lock conversions:
  - Growing Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can **convert** a lock-S to a lock-X (**upgrade**)
  - Shrinking Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (**downgrade**)
- This protocol ensures serializability

Automatic Acquisition of Locks

- A transaction T_i issues the standard read/write instruction, without explicit locking calls.
- The operation **read**(*D*) is processed as:

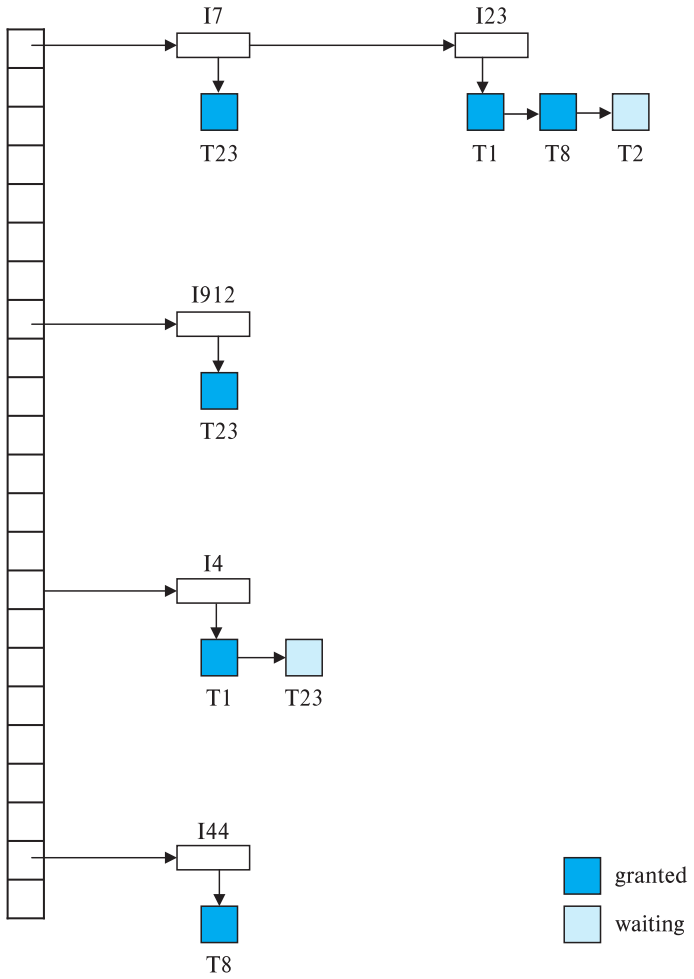
```
if T_i has a lock on D
        then   read(D)
else begin
        if necessary wait until no other transaction has a lock-X on D
        grant T_i a  lock-S on D;
        read(D)
end
```

Automatic Acquisition of Locks

- **All locks are released after commit or abort**
- The operation **write***(D)* is processed as:

```
if Ti has a  lock-X on D
        then  write(D)
else begin
        if necessary wait until no other trans. has any lock on D,
        if Ti has a lock-S on D
                then upgrade lock on D  to lock-X
        else grant Ti a lock-X on D
                write(D)
end;
```

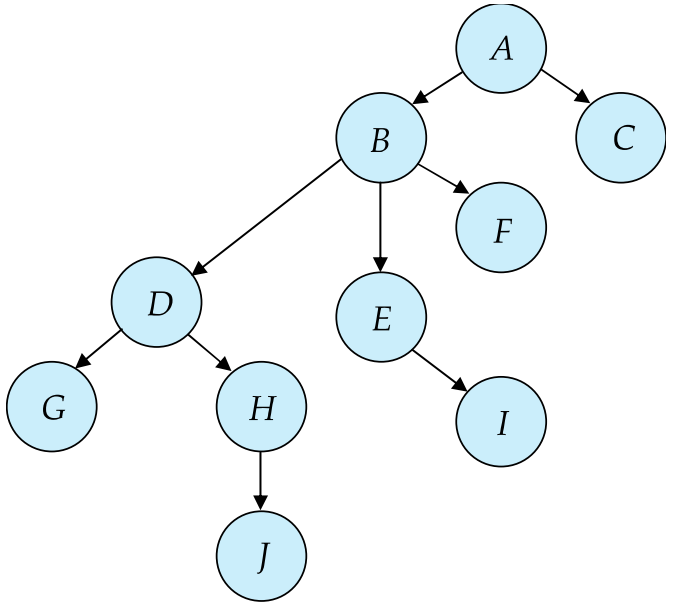
Lock Table



- The lock manager maintains an in-memory data-structure called a lock table to record granted locks and pending requests
  - Txns can send lock and unlock requests to LM
  - The LM replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The LM may keep a list of locks held by each transaction

Graph-Based Protocols

- An alternative to two-phase locking
- Impose a partial ordering on the dataset
- The tree-protocol is a simple kind of graph protocol
  - Ensures conflict serializability as well as freedom from deadlock
  - Unlocking may occur earlier
  - Does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies
  - Have to lock data items that they do not access



Dealing with Deadlocks

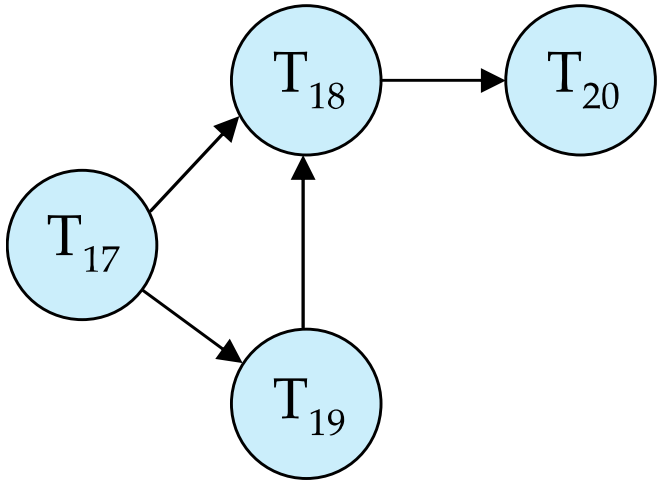
Approach #1: Deadlock Detection

- The DBMS creates a **waits-for graph** to keep track of what locks each txn is waiting to acquire:
  - Nodes are transactions
  - Edge from T_i to T_j if T_i is waiting for T_j to release a lock.
- The system periodically checks for **cycles** in waits-for graph and then decides how to break it.

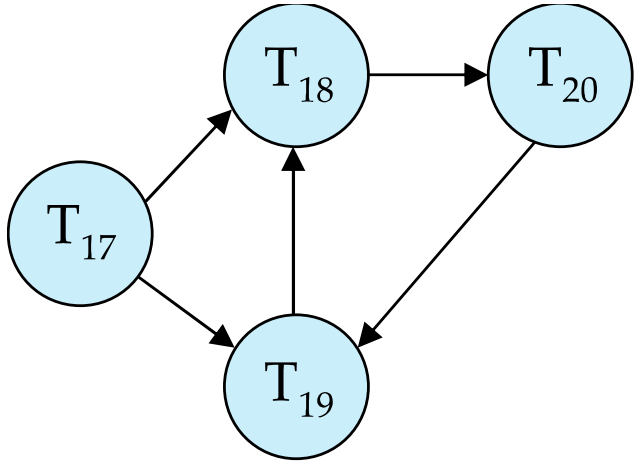
Approach #2: Deadlock Prevention

- When a txn tries to acquire a lock that is held by another txn, the DBMS **kills** one of them to prevent a deadlock.

Deadlock Detection



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Prevention

- **wait-die** scheme
  - Older transaction may wait for younger one to release data item.
  - Younger transactions never wait for older ones; they are rolled back instead.
  - A transaction may die several times before acquiring a lock
- **wound-wait** scheme
  - Older transaction forces younger transaction rollback of instead of waiting
  - Younger transactions may wait for older ones.
  - Fewer rollbacks than *wait-die* scheme.
- In both schemes, a rolled back transactions is restarted with its original timestamp.

Timeout-Based Schemes

- A transaction waits for a lock only for a specified amount of time.
  - After that, the wait times out and the transaction is rolled back.
- Ensures that deadlocks get resolved by timeout if they occur
- Simple to implement
- But may roll back transaction unnecessarily in absence of deadlock
- Difficult to determine good value of the timeout interval.
- Starvation is also possible

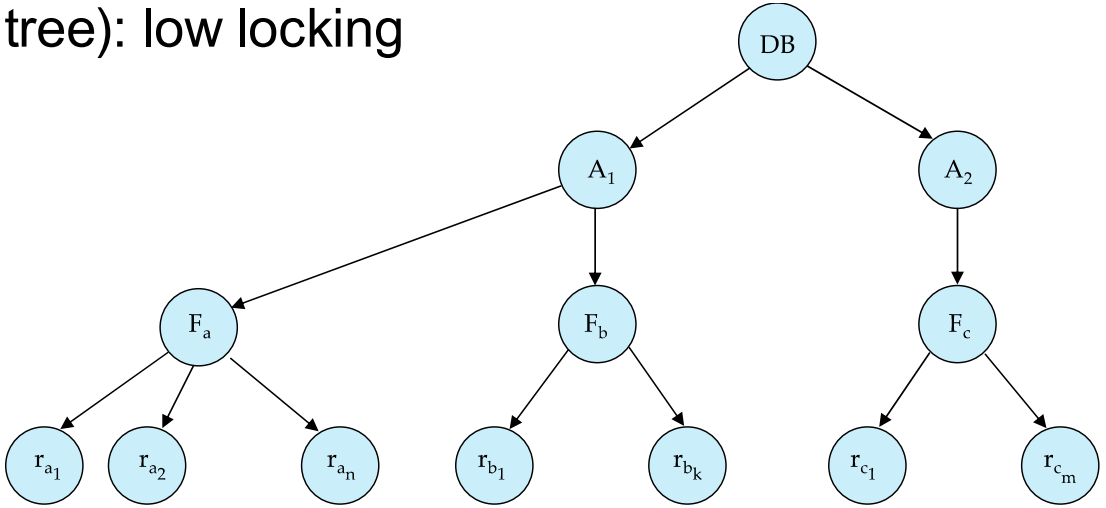
Deadlock Recovery

- Some transaction will have to rolled back (victim) to break deadlock cycle.
  - Select that transaction as victim that will incur minimum cost
- Rollback -- determine how far to roll back transaction
  - **Total rollback**: Abort the transaction and then restart it.
  - **Partial rollback**: Roll back victim transaction only as far as necessary to release locks that another transaction in cycle is waiting for
- Starvation can happen (why?)
  - One solution: oldest transaction in the deadlock set is never chosen as victim

Multiple Granularity

- Granularity of locking (level in tree where locking is done):
  - **Fine granularity** (lower in tree): high concurrency, high locking overhead
  - **Coarse granularity** (higher in tree): low locking overhead, low concurrency

- *database*
- *area*
- *file*
- *record*



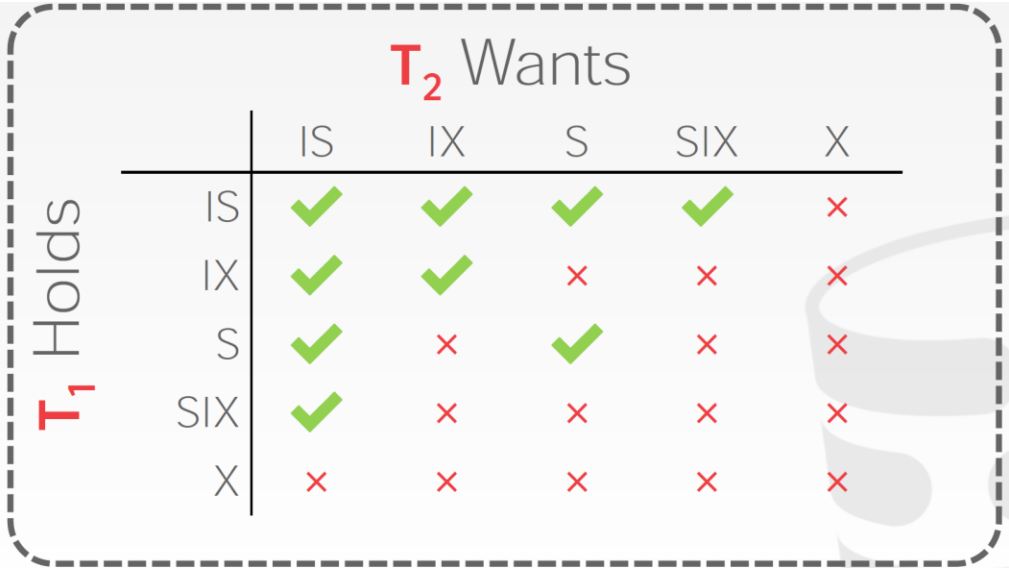
Lock Granularity

- Allow data items to be of various sizes and define a **hierarchy** of data granularities, where the small granularities are nested within larger ones

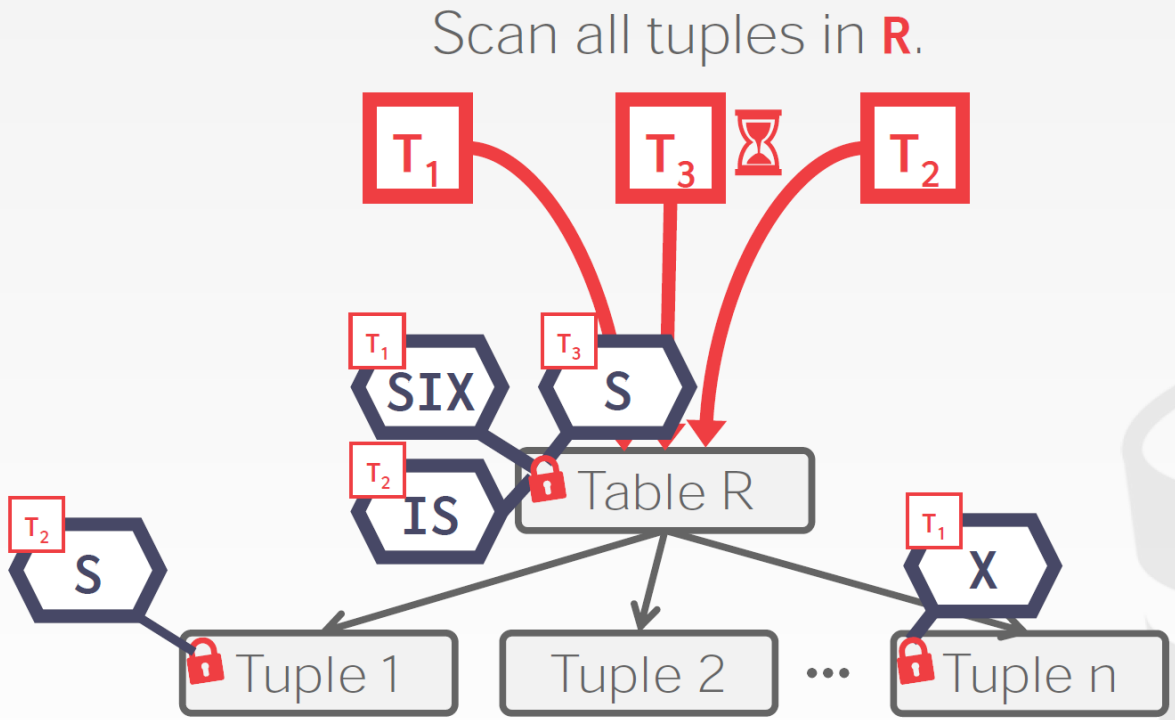
Intention-Shared (IS): Indicates explicit locking at a lower level with shared locks.

Intention-Exclusive (IX): Indicates locking at lower level with exclusive or shared locks.

Shared+Intention-Exclusive (SIX): The subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.



Example


Scan all tuples in R.
T1
T3
T2
T1
SIX
T3
S
T2
IS
Table R
T2
S
T1
X
Tuple 1
Tuple 2
Tuple n

Insert/Delete Operations

- Locking rules for insert/delete operations
  - An **X-mode** lock must be obtained on an item before it is deleted
  - A transaction that inserts a new tuple into the database automatically given an **X-mode** lock on the tuple
- Ensures that
  - reads/writes conflict with deletes
  - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits

Predicate Reads

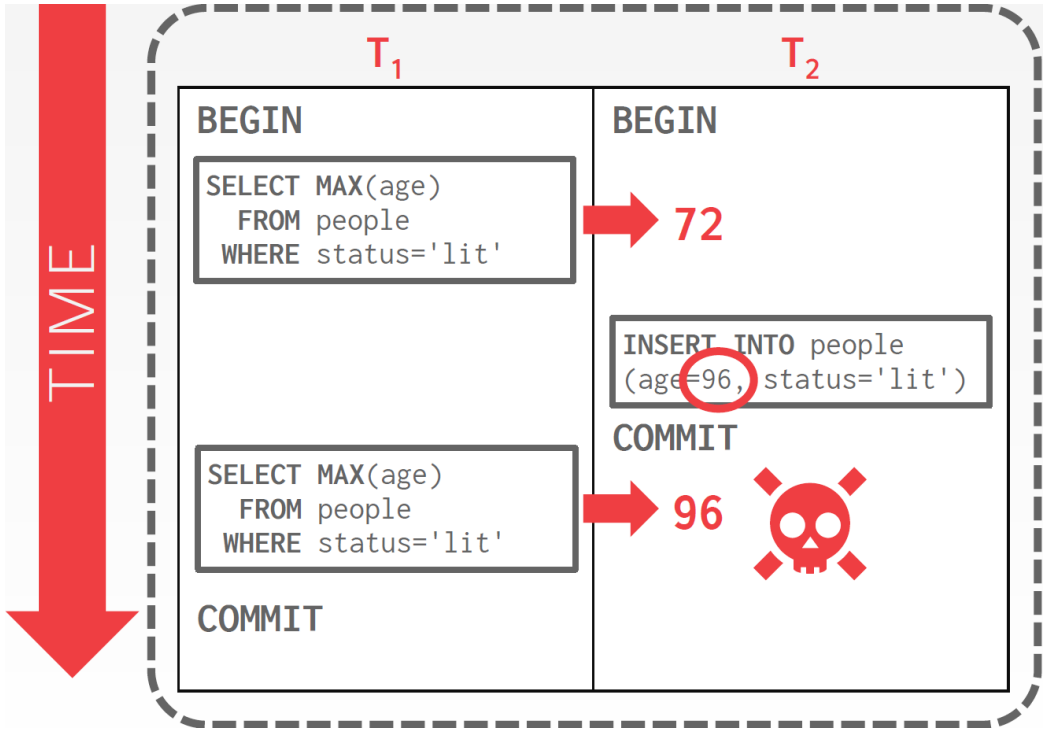
T1	T2
Read(instructor where dept_name='Physics')	
	Insert Instructor in Physics
	Insert Instructor in Comp. Sci.
	Commit
Read(instructor where dept_name='Comp. Sci.')	

Handling Phantoms

- There is a conflict at the database level
  - Predicate read or scanning the relation is reading information that indicates what tuples the relation contains
  - The transaction inserting/deleting/updating a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.
- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)
- Above protocol provides very **low** concurrency for insertions/deletions.

Predicate Locking

- Conflict serializability on reads and writes of individual items guarantees serializability **only** if the set of objects is fixed.
- Lock records that satisfy a logical predicate:
  - Example: status='lit'
- In general, predicate locking has a lot of locking overhead.
- **Index locking** is a special case of predicate locking that is potentially more efficient.

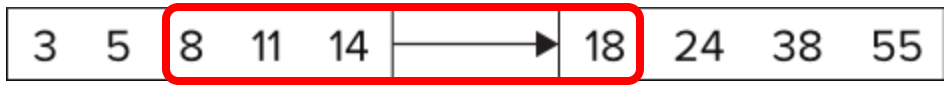


Index Locking

- Every relation must have at least one index.
- A transaction can access tuples only after finding them through one or more indices on the relation
- A transaction T_i that performs a lookup must lock all the index **leaf nodes** that it accesses, in S-mode
  - Even if the leaf node does not contain any tuple satisfying the index lookup
- A transaction T_i that inserts, updates or deletes a tuple t_i in a relation **r**
  - Must update all indices to **r**
  - Must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
- The rules of the two-phase locking protocol must be observed

Next-Key Locking

- Index-locking protocol can result in **poor** concurrency if there are many inserts
- Next-key locking protocol: provides higher concurrency
  - Lock all values that satisfy index lookup
  - Also lock next key value in index
    - even for inserts/deletes
  - Consider B+-tree leaf nodes as below, with query predicate $7 \leq X \leq 16$. Check what happens with next-key locking when inserting: (i) 15 and (ii) 7



Concurrency in Index Structures

- Indices‘ job is to help in accessing data.
- Index-structures are typically accessed very often.
  - 2-phase locking of index nodes can lead to low concurrency.
- Index concurrency protocols
  - Locks on internal nodes are released early, and not in a two-phase fashion.
  - It is acceptable to have non-serializable concurrent access to an index
    - The accuracy of the index is maintained.
    - In particular, the exact values read in an internal node of a B^+-tree are irrelevant so long as we land up in the correct leaf node.

Crabbing Protocol

- Instead of two-phase locking on the nodes of the B$^+$-tree
  - First lock the root node in **shared** mode.
  - After locking all required children of a node in shared mode, release the lock on the node
  - During insertion/deletion, upgrade leaf node locks to exclusive mode.
  - When splitting or coalescing requires changes to a parent, lock the parent in exclusive mode.
- Could cause excessive deadlocks
  - Can abort and restart search, without affecting transaction