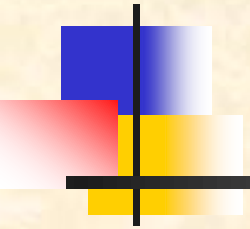


数据库系统原理

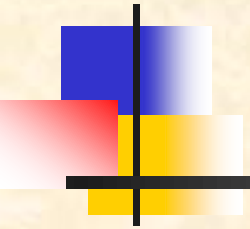
Database System Principle

邵蓥侠

Email：shaoyx@bupt.edu.cn

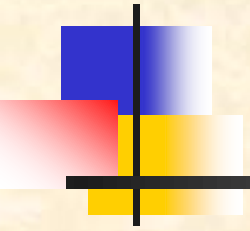
北京邮电大学计算机学院

计算机应用技术中心



PART 7

TRANSACTION MANAGEMENT



Chapter 18

Concurrency Control

Introduction to Chapter 18

- Concurrency control of processes
  - 多个进程对共享资源的使用：互斥使用
  - OS中的进程并发控制（互斥与同步）机制：
    软件方法，硬件方法，信号量，管程
  - 信号量方法：　wait(S)
    critical section
    signal(S)
  - 锁/lock S：二元信号量，实现对共享资源的互斥使用，{0，1} 表示资源使用情况，锁操作控制进程的执行和等待
  - deadlock handling
    - deadlock prevention, deadlock avoidance, deadlock detection and recovery

Introduction to Chapter 18 (cont.)

- Transaction concurrency control
  - /*根据可串行化基本原理, 采用以下机制控制多个进程对<u>共享数据</u>的<u>互斥访问</u>, 实现对事务的并发调度
    - lock-based protocols (§ 18.1)
    - multi- granularity schemes (多粒度， § 18.3)
    - {multi-version schemes (多版本， § 18.6)
    - timestamp-based protocols (§ 18.4)

      validation-based protocols (验证， § 18.5) }
    - deadlock handling (§ 18.2)

      deadlock prevention

      deadlock detection and recovery

§ 18.1 Lock-based Protocol

- Principles

  /*用二元信号量 “lock”控制并发事务{T_i}对共享数据Q的访问

  - T_i存取Q时，先向DBMS申请对Q加锁
  - 如果此时Q未被其它事务加锁，则T_i加锁成功，继续执行对Q的访问
  - 如果此时Q已经被其它事务加锁，则T_i等待，直至Q上的锁被释放， T_i获得锁后继续执行
  - T_i执行完对Q的操作后，解锁Q
  - DBMS根据lock-*compatibility* matrix将lock授予请求的事务
  - 事务执行过程： 申请锁——加锁——读写——解锁

Lock-based Protocol

- 共享数据Q
  - 1.逻辑单元：

    属性值，属性值集合，元组，关系，索引项，整个索引，数据库DB
  - 2. 物理单元：数据页，索引页，数据块extent

18.1-1 Locks

- A lock is a mechanism to control concurrent access to a data item
  - a transaction is allowed to access a data item only if it is currently holding a lock at that item
- Data items can be locked in two modes :
  - **exclusive (X) mode**. Data item can be both read as well as written. X-lock is requested using lock-X instruction
  - **shared (S) mode**. Data item can only be read. S-lock is requested using lock-S instruction
- Lock requests are made to the concurrency-control manager by the programmer. Transaction can proceed only after request is granted.
- Lock primitives
  - lock-S(Q), unlock-S(Q); lock-X(Q), unlock-X(Q);

Locks (cont.)

- Concurrency-control manager/scheduler in DBMS grants lock to transactions who want to access Q according to *lock-compatibility* matrix
  - assuming that T_i requests a lock of *mode **A*** on data item Q on which another transaction T_j currently holds a lock of *mode **B***, if T_i can be granted a lock of *mode A*, then *mode **A** is compatible with mode **B***

T_i \ T_j	S	X
S	true	false
X	false	false

Fig.18.1 Lock-compatibility matrix

Locks (cont.)

- In accordance with Fig.18.1, several shared locks can be held simultaneously (by different transactions) on shared data item Q;
- When T_i requests a lock of *mode **A*** to operate on data item Q, e.g. S(***Q***) related to ***read(Q)***, while Q is already locked in a incompatible *mode **B*** byT_j, e.g. X***(Q)*** related to ***write(Q)***, then
  - T_i must wait until the lock in *mode **B*** held by T_j is released, the lock is then granted to T_i
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted

18.1-1 Locks (cont.)

- Procedures of T_i's executing to access a data item Q

  (1) other operations in T_i

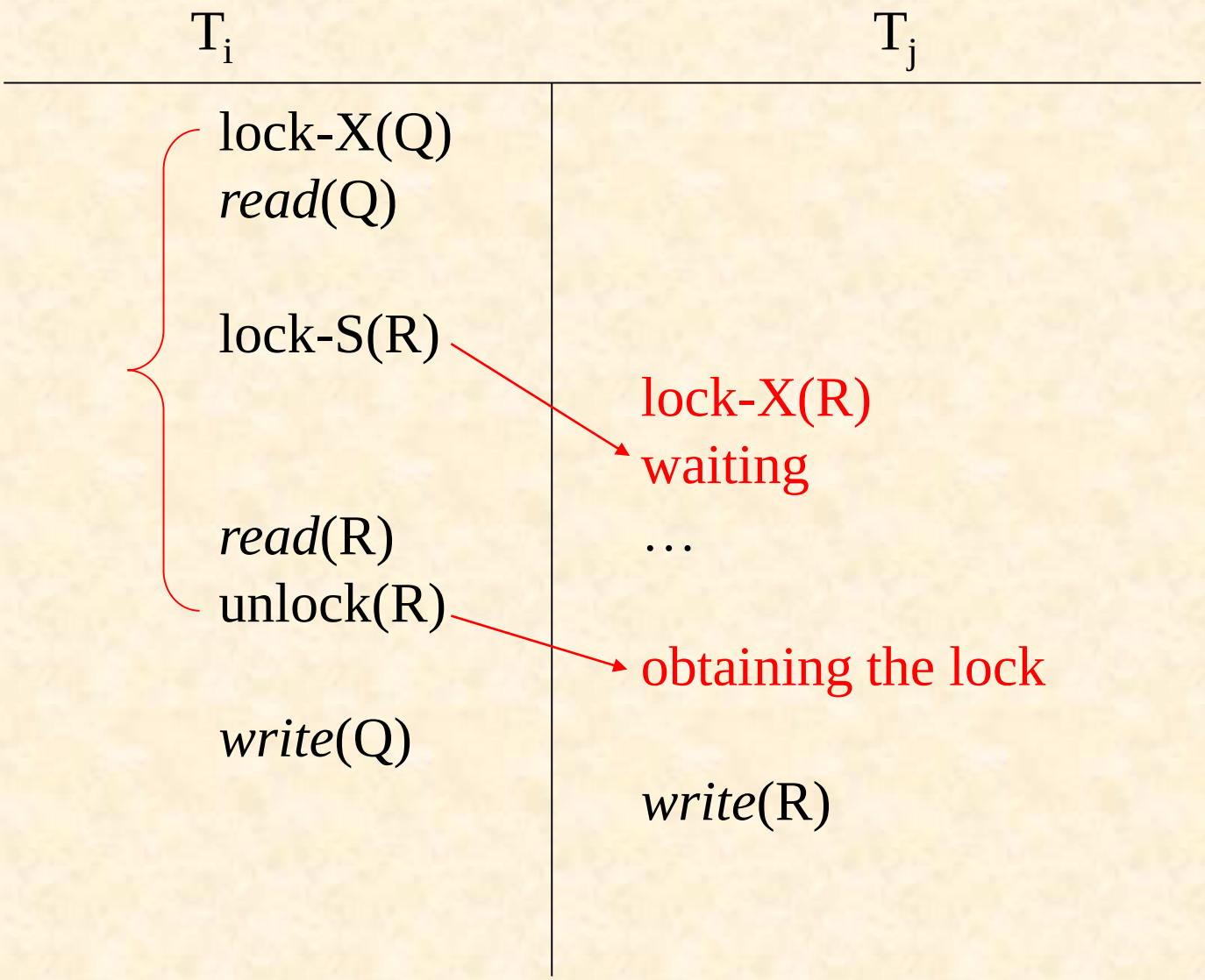
  (2) ***request*** a lock S(Q) or X(Q), by the primitive lock-S(Q) or lock-X(Q)

  (3) ***wait for*** concurrency-control manager's granting of the lock requested;

  (4) whenever obtaining the lock, make ***access*** on Q;

  (5) after finishing data access on Q, ***release*** lock on Q, by unlock-S(Q) or unlock-X(Q);

  (6) other operations;


T_i
T_j
lock-X(Q)
read(Q)
lock-S(R)
lock-X(R)
waiting
…
read(R)
unlock(R)
obtaining the lock
write(Q)
write(R)

AntiExamples

- T_1: transfer $50 from account-B to account-A, Fig.18.2
- T_2: display the total amount of A and B, Fig.18.3
- T_1 and T_2 ***unlock*** data item A and B immediately after its final access of that data
- Initially, A=100, B=200

```
T1:   lock-X(B);
      read(B);
      B:=B-50;
      write(B);
      unlock(B);
      lock-X(A);
      read(A);
      A:=A+50;
      write(A);
      unlock(A);
```

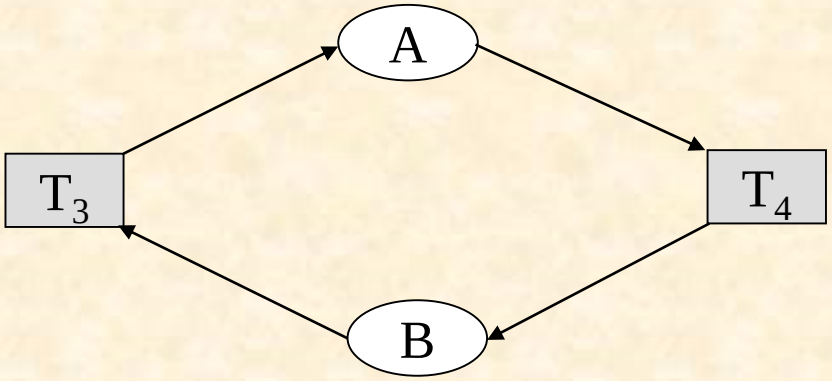
```
T2: lock-S(A);
      unlock(A);
      lock-S(B);
      read(B);
      unlock(B);
      display(A+B)
```

- For the concurrent schedule S1 (given in Fig.18.4) on T_1 and T_2 , an *inconsistent* result **display(A+B) = 250** exist, why?
  - T_1 unlocked data item B too early, as a result of which T_2 saw a inconsistent state

T_1	T_2	concurreny-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_1)
read(A)		
$A := A - 50$		
write(A)		
unlock(A)		

AntiExamples (cont.)

- T_3: transfer $50 from account-B to account-A, Fig.18.5
- T_4: display the total amount of A and B, Fig.18.6
- T_3 and T_4 ***unlock*** data item A and B at the end of transaction
- Initially, A=100, B=200
- The concurrent schedule S2 (given in Fig.18.7) on T_3 and T_4 leads to ***deadlock***



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)	

Locking protocol

- As revealed by the examples, in lock-based schemes, the orders of granting locks to and revoking locks from transactions must be limited to avoid inconsistent DB states and starvation of transactions

- Locking protocol
  - set of rules followed by all transactions while requesting and releasing locks
- Locking protocols restrict the set of possible schedules, and lead to a subset of all possible conflict serializable schedules

18.1-3 Two Phase Locking Protocol/2PL

- This protocol ensures conflict-serializable schedules
- In accordance with 2PL, each transaction T_i issues ***lock*** and ***unlock*** requests in two phases
  - growing phase: T_i may obtain locks, but may not release any locks
  - shrinking phase:T_i may release locks, but may not obtain any locks
  - when T_i begins, it enters growing phase
  - when T_i releases its ***first lock*** by ***unlock***, it enters shrinking phase, and is not allowed to apply for locks, i.e. not allowed to enter growing phase

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

transaction following 2PL

T_2: **lock-S**(*A*);
read (*A*);
unlock(*A*);
lock-S(*B*);
read (*B*);
unlock(*B*);
display(*A*+*B*)

transaction
not following

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure 15.8 Partial schedule under two-phase locking.

Two Phase Locking Protocol (cont.)

- The protocol assures 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).

- **Lock point**
  - the point in the transaction/schedule where the transactions has obtain its final lock, but has not yet unlock any locks

- E.g. lock-point in T_5 in Fig.18.0.1

Two Phase Locking Protocol (cont.)

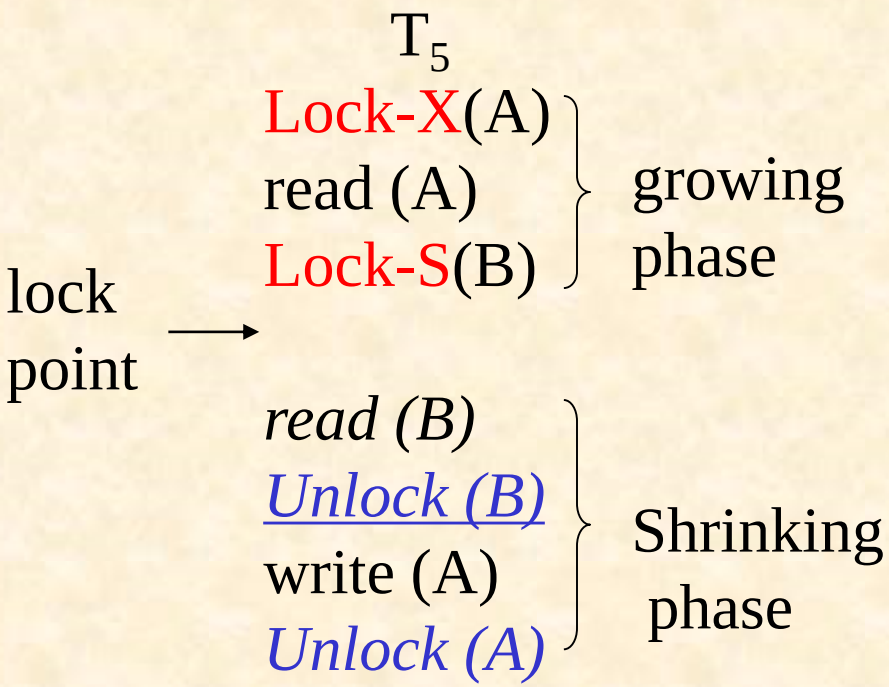


Fig.18.0.1 Partial schedule under two-phase locking

!! Properties of 2PL

- 2PL guarantees ***conflict serializability***, i.e. generates conflict serializable concurrent schedules

- 2PL does not ensure freedom from deadlock
- 2PL does not avoid cascading rollback

Two Phase Locking Protocol (cont.)

- Given a concurrent schedule S in line with 2PL, S is equivalent to a serial schedule S', such that
  - the ordering of lock points of each transactions in S is the same as the ordering of transactions in S'

- E.g. S3 in Fig.18.0.1 is conflict serializable, and equivalent to $<T_5; T_6'>$
  - the lock point of T_5 is before the lock point of T_6'

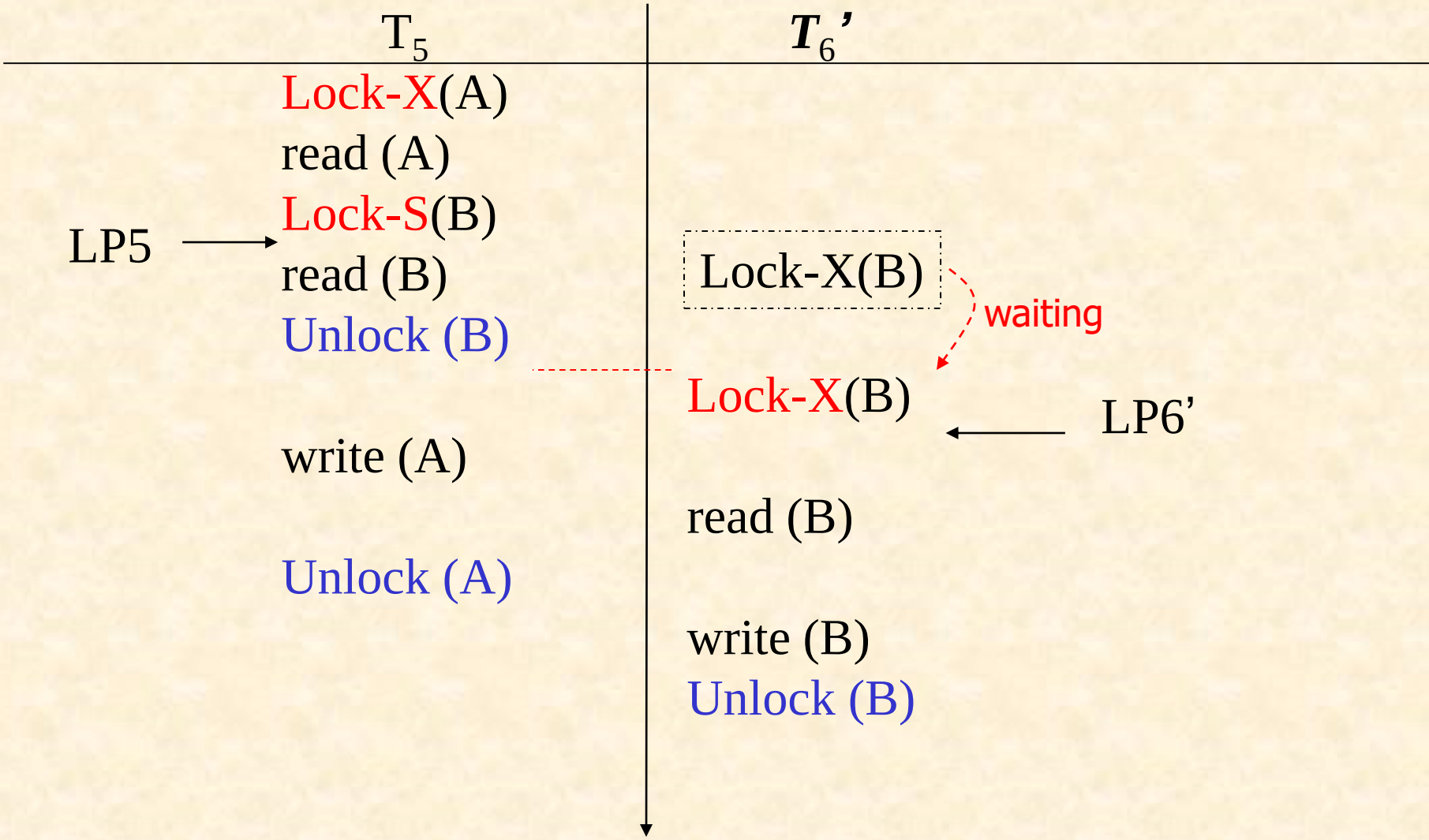
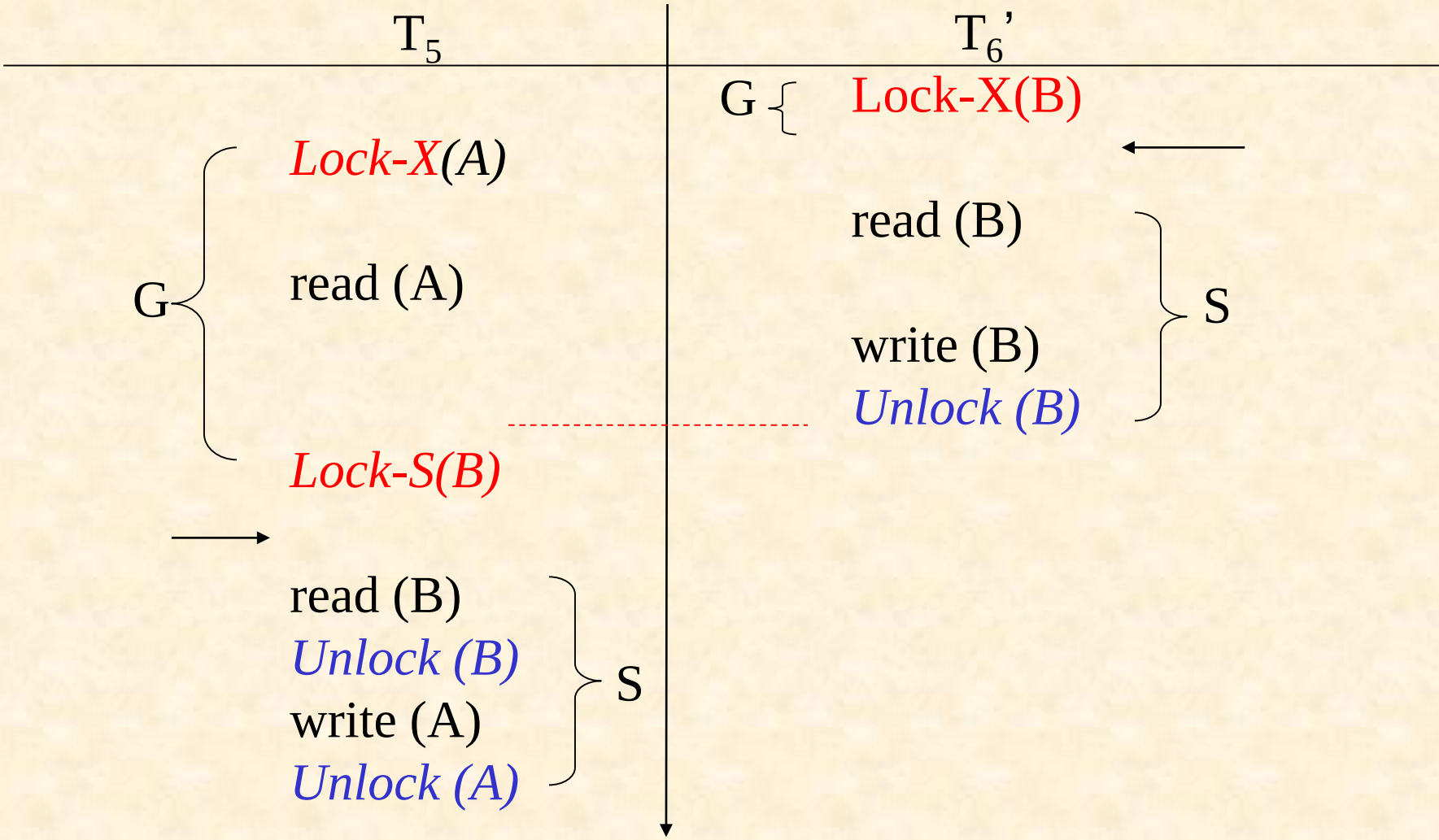


Fig. 18.0.1 A concurrent schedule S3 under 2PL protocol constraints

How to construct the schedules in line with 2PL

- /*给定若干事务, 构造满足2PL的并发调度S！！
  - 为各事务添加加锁和解锁操作，各事务应满足2PL要求，即事务内部分为生长和收缩2个阶段
  - 根据各事务的锁点位置和锁的互斥性 (相容关系)，安排各事务的并发执行
  - S中各事务的growing phase和shrinking phase 应尽可能错开，不允许访问同一数据的2个事务（冲突事务）同时处于growing phase

- E.g. A concurrent schedule S4 in Fig.18.0.2



- S4 is equivalent to <T6';T5>

Fig. 18.0.2 A concurrent schedule S4 under 2PL protocol constraints

How to construct the schedules in line with 2PL (cont.)

- E.g. Construct a concurrent schedule that is equivalent to < T7;T8 >
  - step1. 分别为T7和T8加锁操作, 构成T7'和T8'

T_7	T_7'	T_8	T_8'
read (A)	Lock-X(A)	read (B)	Lock-X(B)
read (B)	read (A)	write (B)	read (B)
write (A)	Lock-S(B)	read (C)	Lock-S(C)
	read (B)		write (B)
	Unlock (B)		read (C)
	write (A)		Unlock (B)
	Unlock (A)		Unlock (C)

Fig.18.0.3-1 A concurrent schedule S5

- step2. 根据锁的相容性构造调度，假设T_7'的锁点先于T_8'的锁点

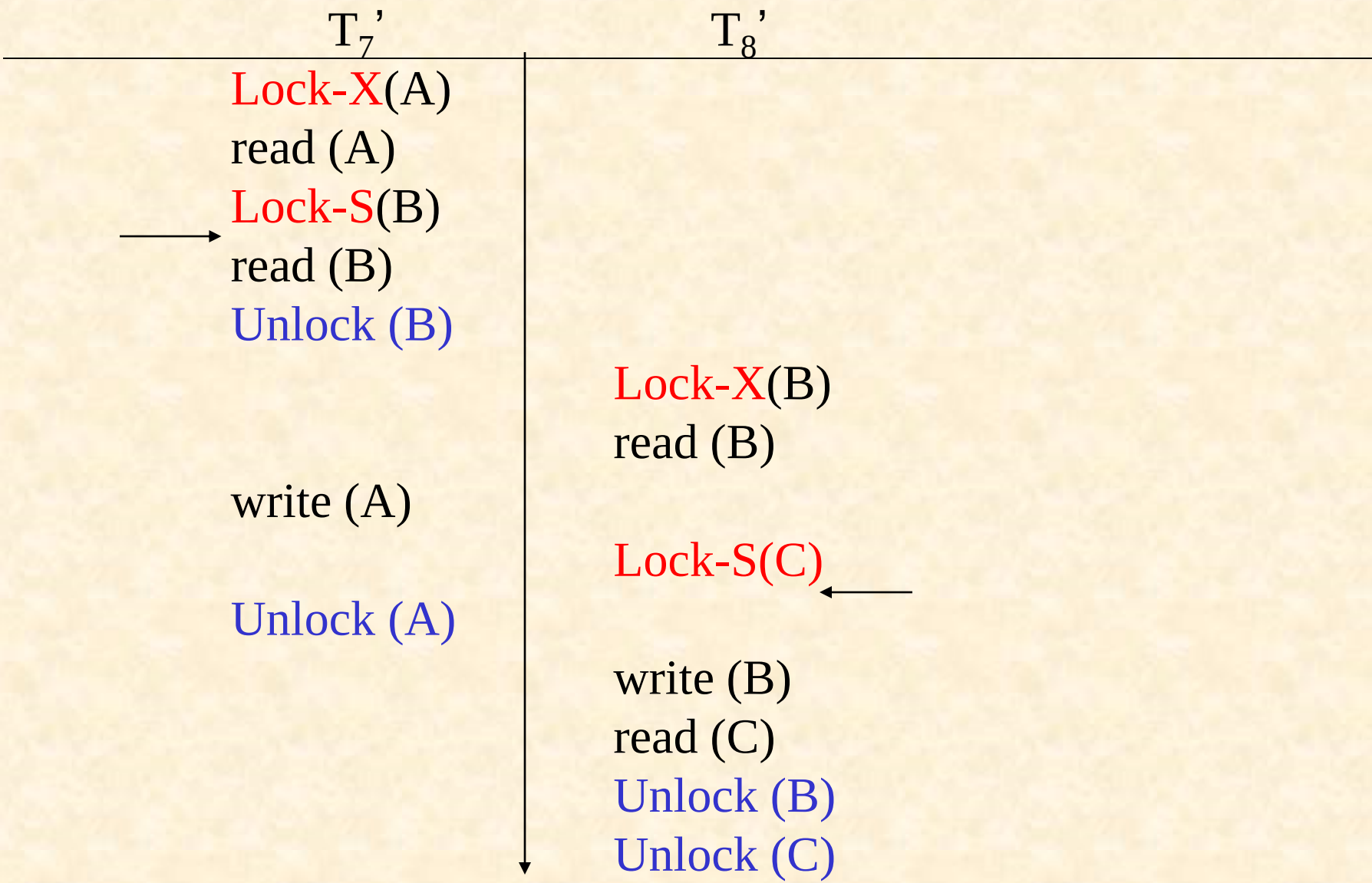


Fig.18.0.3-2 A concurrent schedule S5

Strict 2PL protocol

- There are two types of enhanced 2PL protocols, that is, strict 2PL and rigorous 2PL

- Strict (严格) 2PL protocol
  - all X-locks taken by a transaction are held until that transaction commits, that is, all X-locks can only be unlocked at the end of the transaction
  - leading to ***cascadeless*** and ***serializable*** schedules

- E.g. schedule S6 in Fig.18.0.4

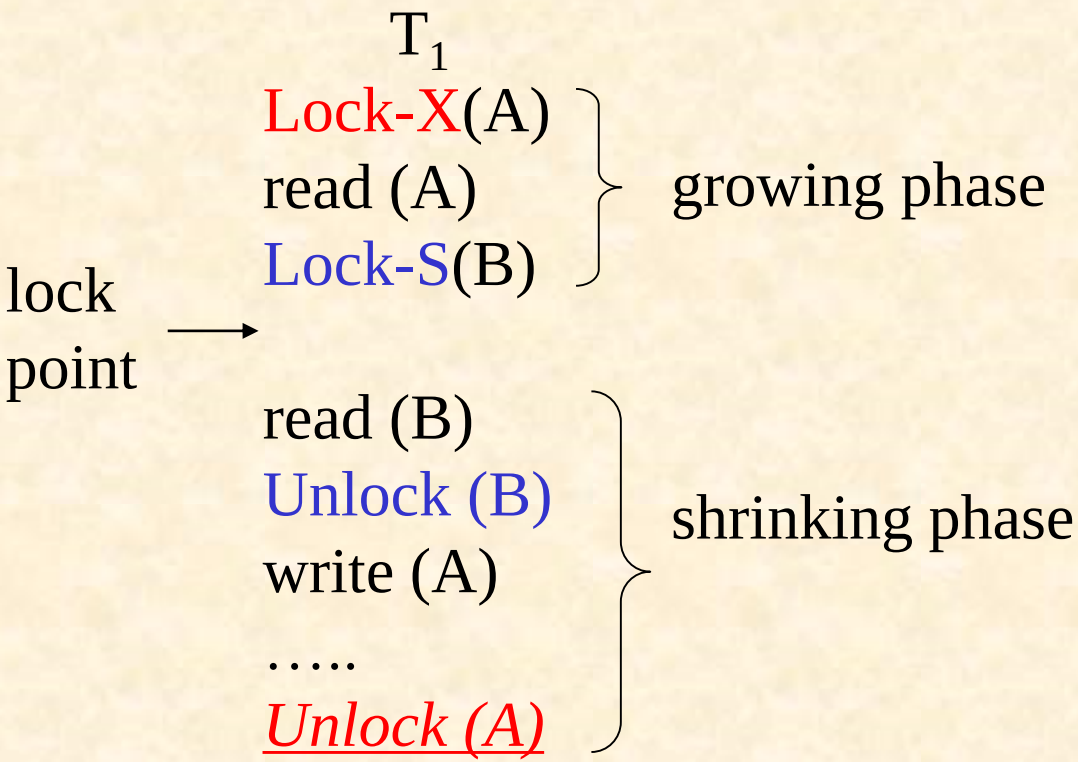


Fig.18.0.4 A schedule S6 under strict 2PL

Rigorous 2PL protocol

- Rigorous (强制) 2PL protocol
  - all X-locks and S-locks taken by a transaction are held until that transaction commits
    - leading to transactions are serialized in the order in which they commit
- Schedule S7 in Fig.18.0.5
- Most DBMS implement either strict or rigorous 2PL protocol.

Rigorous 2PL protocol (cont.)

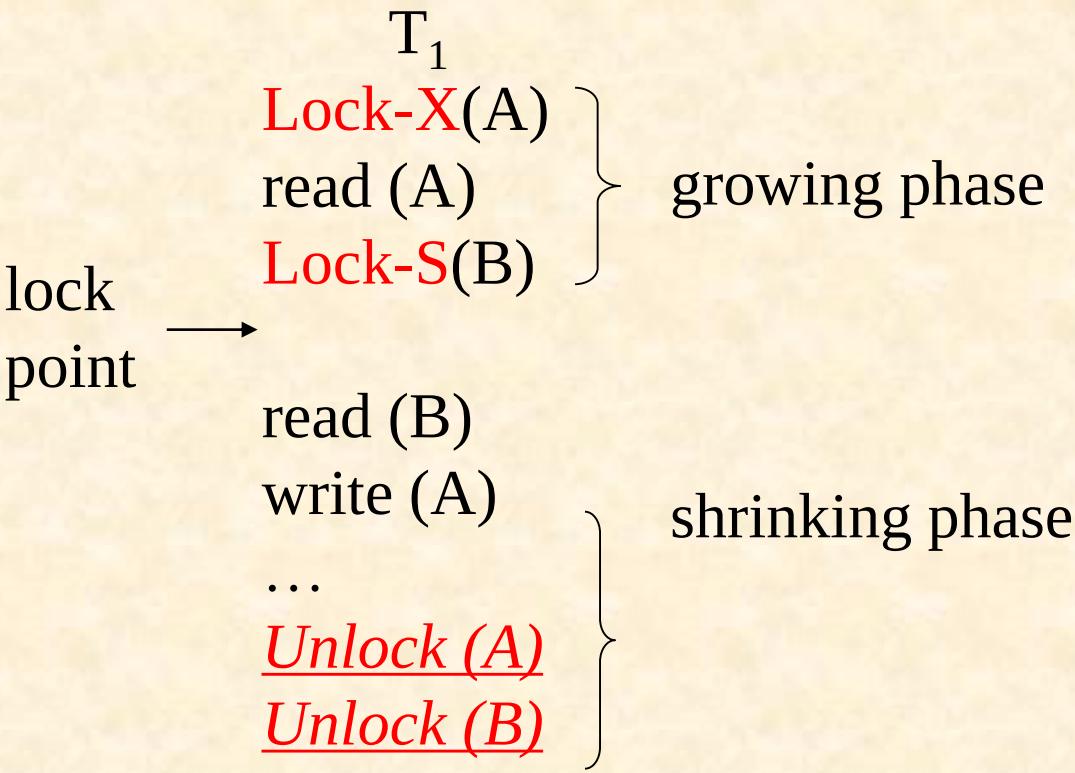
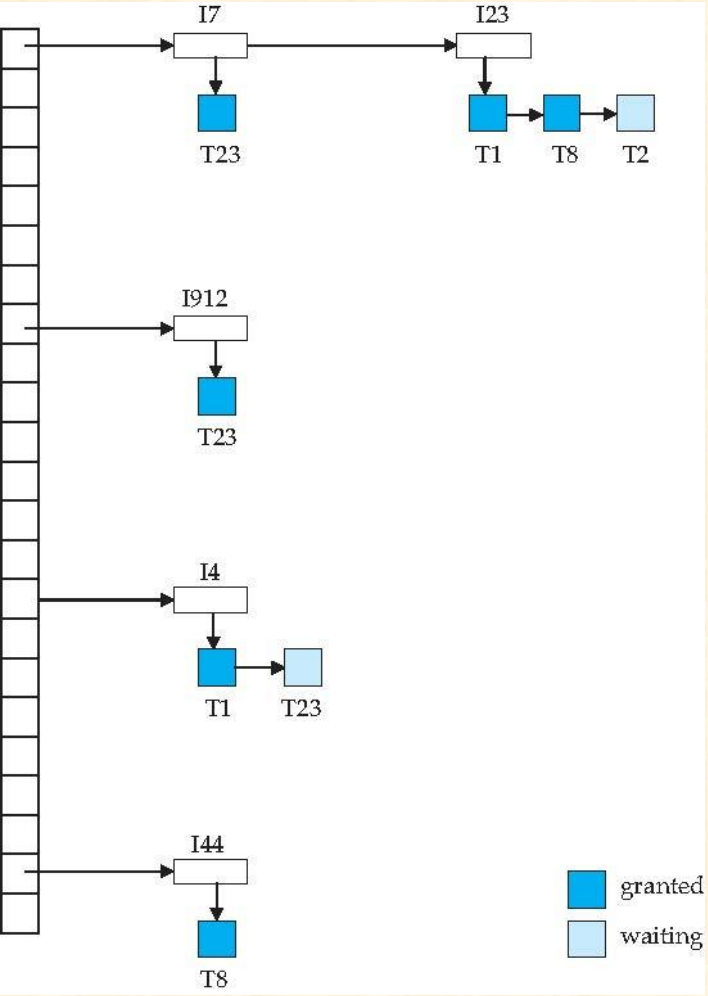


Fig.18.0.5 A schedule S7 under rigorous 2PL

18.1.4 Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager 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 requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



- Dark blue rectangles indicate granted locks; light blue indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Deadlock Handling

System is **deadlocked** if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set,and none of the transactions can ever proceed with its normal executions.

T_1	T_2
lock-X on *X* write (*X*)	
	lock-X on *Y* write (*X*) wait for **lock-X** on *X*
wait for **lock-X** on *Y*	

Deadlock Prevention Strategies

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state.

★Require that each transaction locks all its data items before it begins execution (predeclaration).

★Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

★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.

Following schemes use transaction timestamps for the sake of deadlock prevention alone:

- **wait-die** scheme

If TS(Ti) < TS(Tj) then Ti Waits
Else
RollBack Ti
Restart Ti with the same TS(Ti)
Endif

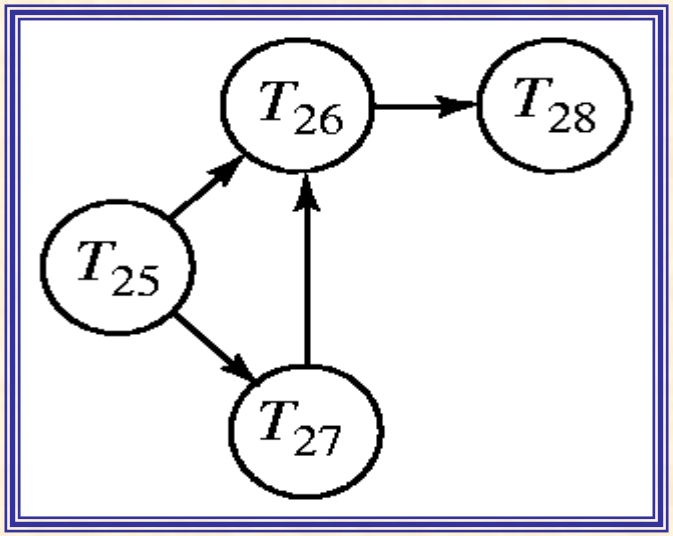
- **wound-wait** scheme

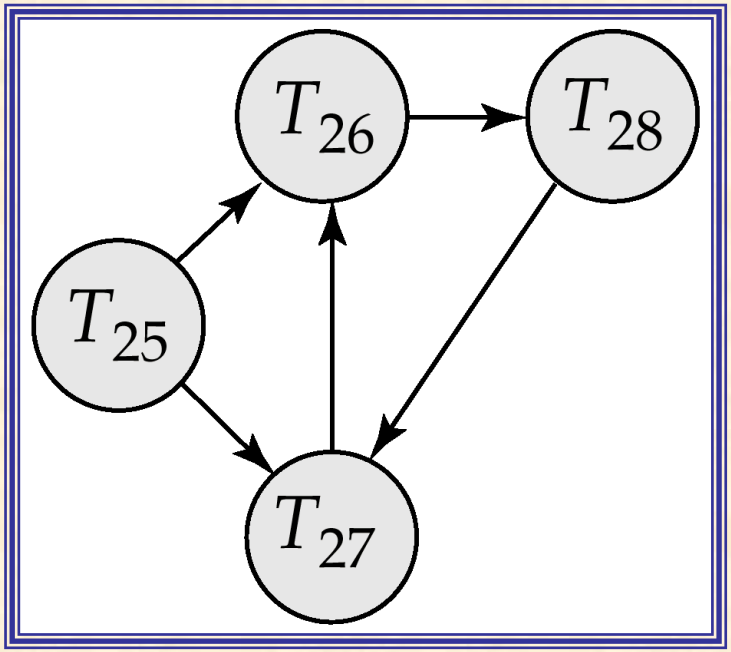
If TS(Ti) > TS(Tj) then Ti Waits
Else
RollBack Tj
Restart Tj with the same TS(Tj)
Endif

- **wait-die** scheme — non-preemptive
  - ★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 needed data item
- **wound-wait** scheme — preemptive
  - ★older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - ★may be fewer rollbacks than *wait-die* scheme.

Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.


T_{26}
T_{28}
T_{25}
T_{27}


T_{26}
T_{28}
T_{25}
T_{27}

Deadlock Recovery

When deadlock is detected :

★Some transaction will have to rolled back (made a victim) to break deadlock. 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.
- More effective to roll back transaction only as far as necessary to break deadlock.

★Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

§ 18.3 Multiple Granularity

- Granularity of data items
  - the size of the data items to be accessed
  - the size of the data items on which synchronization (e.g locking) is performed
- Multiple granularity
  - allow data items, which are accessed by transactions, to be of various sizes
  - define a hierarchy of data granularities, where the small granularities are nested within larger ones

Multiple Granularity (cont.)

- /*在关系数据库中可以加锁的数据对象

  1.逻辑单元：

  属性值、属性值集合、元组、关系、索引项、整个索引、整个数据库

  2. 物理单元：数据页、索引页、数据块, file

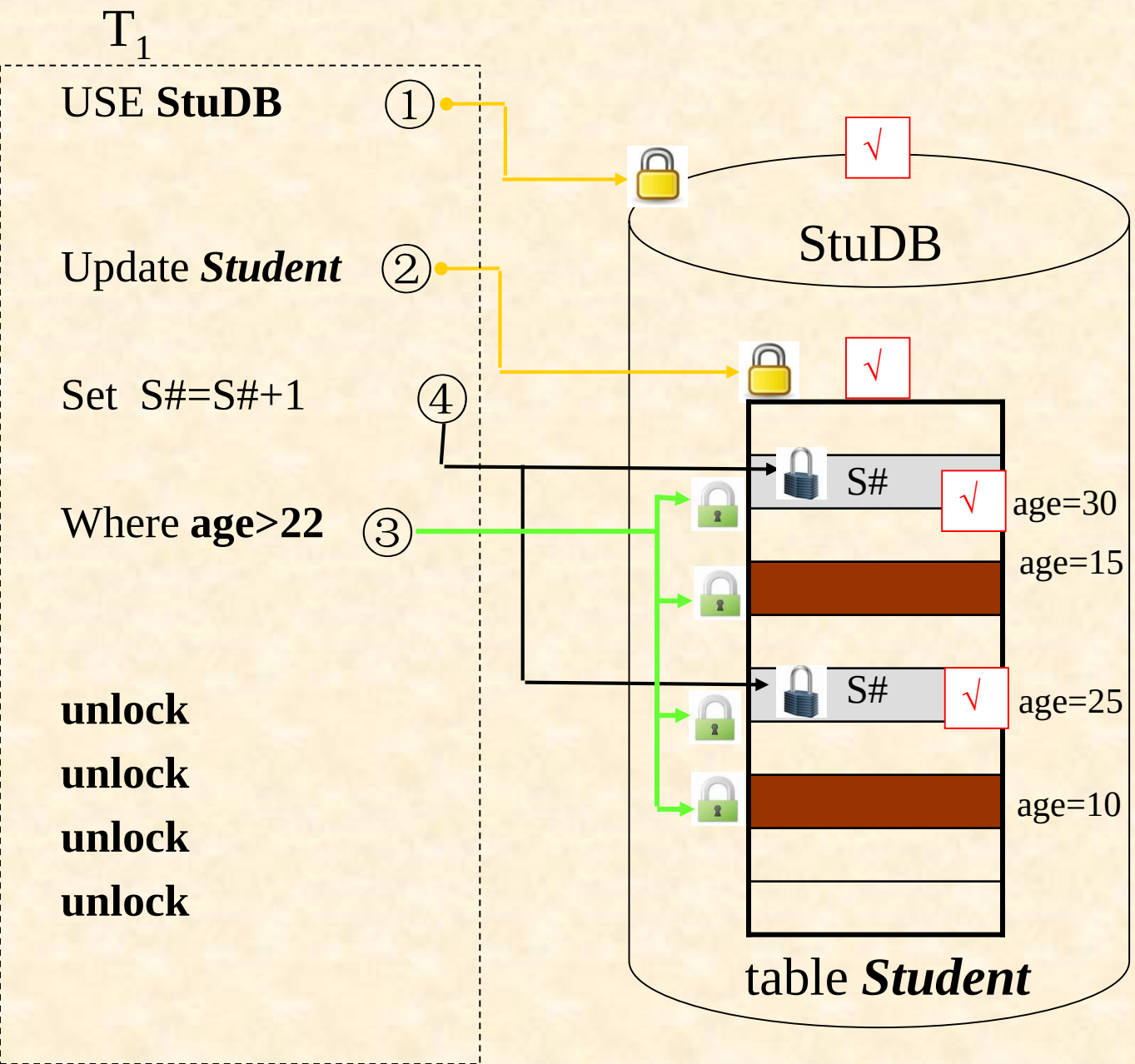
Multiple Granularity (cont.)

- Multiple granularity can be represented graphically as a tree
  - refer to Fig. 18.16 ▶
- Granularity of locking
  - level in tree where locking is done

  — **fine granularity**， lower in tree

  — **coarse granularity,** higher in tree
  - each node in the tree can be locked individually by lock primitives, e.g. *lock_S(Q), lock_X(Q)*
  - **in addition**, when a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node‘s descendents（后代） in the same mode

自上而下四级加锁： DB– table- row- attribute



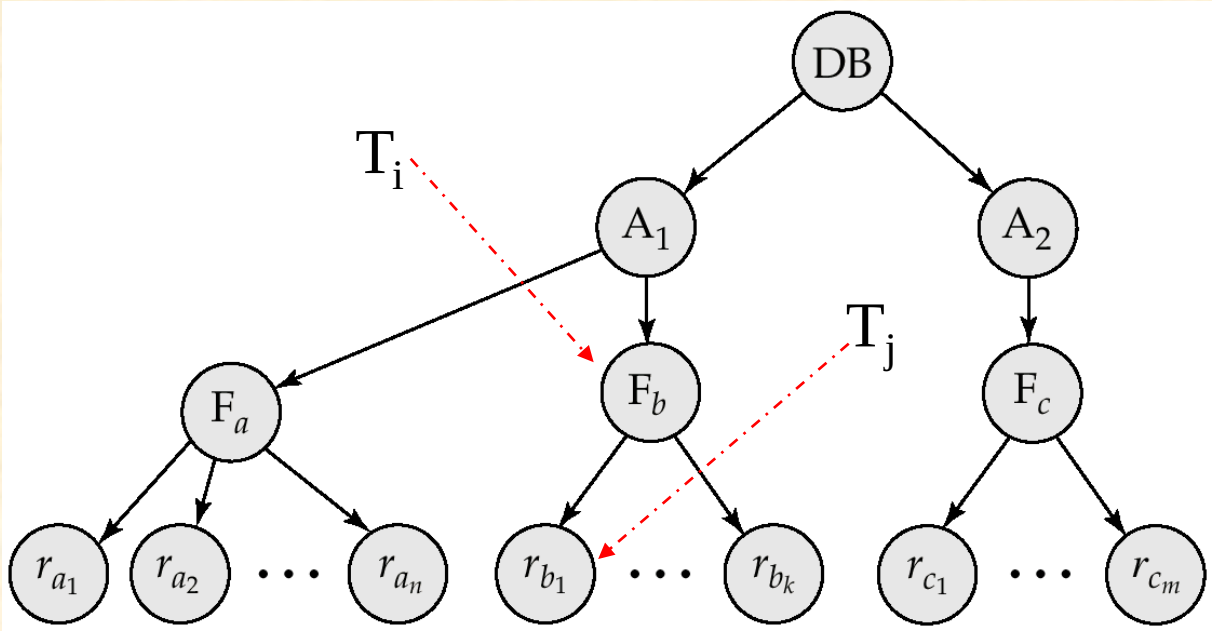


Fig. 18.16 Example of Granularity Hierarchy

- The levels, starting from the coarsest (top) level are
  - *database*
  - *area, e.g. extent, page in SQL Server*
  - *file*
  - *record*

Multiple Granularity (cont.)

- For example, in Fig. 18.16 (▶), T_i has locked F_b in the *exclusive mode* explicitly
  - if T_j wishes to get a *shared* or *exclusive* lock on record r_{b1}, then it has to wait
  - if T_k wishes to lock the entire database, that is, to lock the root node, it will not succeed

Multiple Granularity (cont.)

- Granularity of locking *vs* Concurrency
  - **fine granularity** : high concurrency, high locking overhead
  - **coarse granularity**: low locking overhead, low concurrency
  - /*锁粒度与事务并发执行的程度和 DBMS并发控制的开销密切相关

— 锁粒度越大，系统中可以被加锁的数据项就越少，事务并发执行度也越低，但同时系统开销也小；

— 当锁粒度比较小时，事务并发度高，但系统开销也较大

  - 实际数据库系统中，多个事务并发执行时，DBMS自动为事务访问的数据项选择加锁粒度

Multiple Granularity (cont.)

- 说明：

  从并行效率和管理成本角度，绝大多数数据库系统的最小加锁粒度是行级锁

transaction T_i :

update *Student*
set grade=4
where age>10

transaction T_j :

update *Student*
set grade=6
where age < 5

表级锁

表级锁

行级锁

行级锁

Student

grade	ID	name	age
1	21	1	12
2	22	2	7
3	0	3	8
6	4	3	1
.....			
12	2	7	3
23	4	8	9

Multiple Granularity (cont.)

- 如果对较低级别的某个结点资源加锁，从树根到该结点的直接父节点都必须设置体现出该封锁模式的意向锁，以避免2个事务因封锁不同级别的资源而产生潜在的冲突！！

→意向锁

Intention Lock Modes

- If a node is locked in an intension mode, *explicit* locking is being done at a *lower* level of the tree
  - intension locks are put on all the ancestors of a node *before that node is locked explicitly*

- With respect to the multiple granularity scheme, in addition to *S* and *X* lock modes, there are three additional lock modes
  - ***intention-shared*** (IS):

    if a node is locked in a IS mode, then explicit locking is being done at a lower level of the tree, but only with shared locks

Intention Lock Modes (cont.)

- ***intention-exclusive*** (IX):

  if a node is locked in a IX mode, then explicit locking is being done at a lower level of the tree, but with *exclusive* or *shared* locks

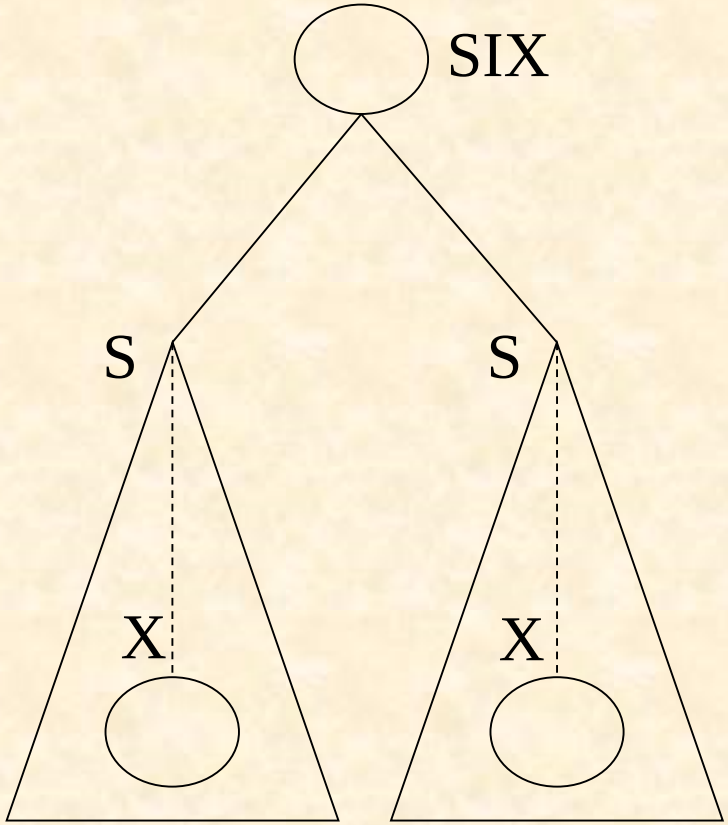
- ***shared and intention-exclusive*** (SIX):

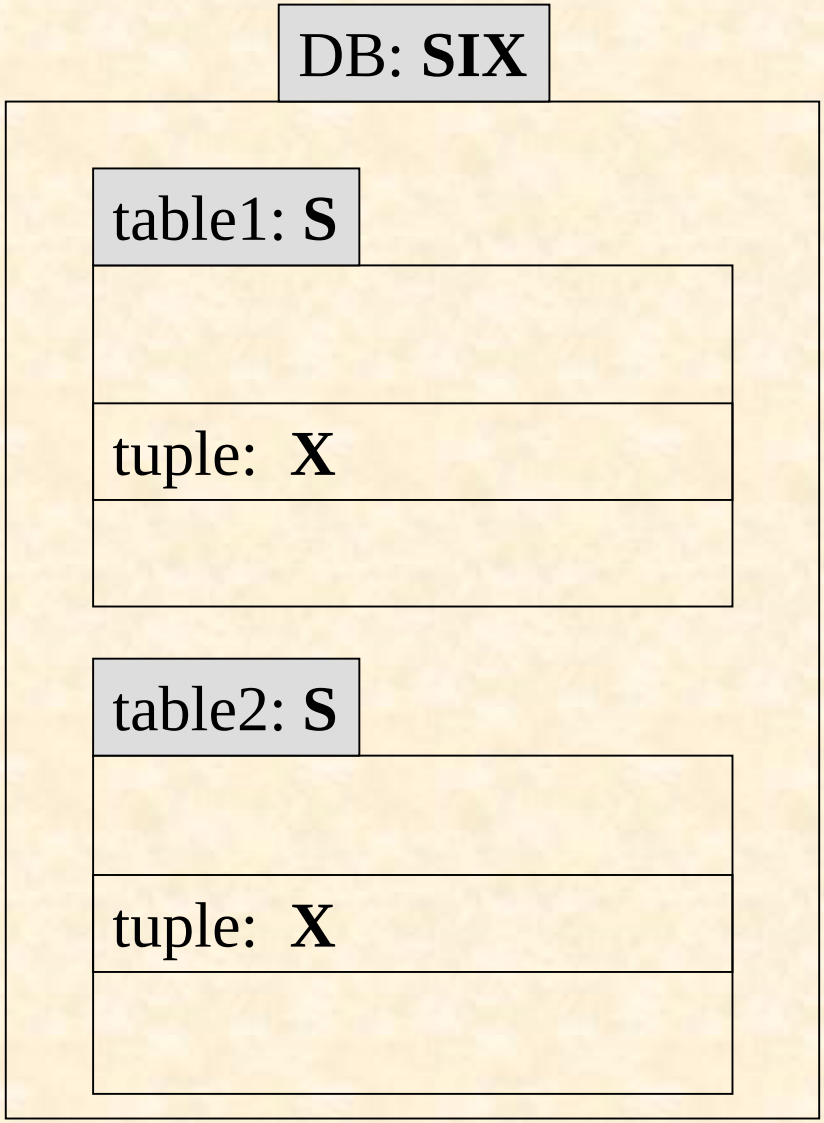
  if a node is locked in a SIX mode,

(1) the *subtree* rooted by that node is locked explicitly in *shared* mode

(2) and explicit locking is being done at *a lower level* with *exclusive-mode* locks.

- Intention locks allow a higher level node to be locked in *S* or *X* mode without having to check all descendent nodes.

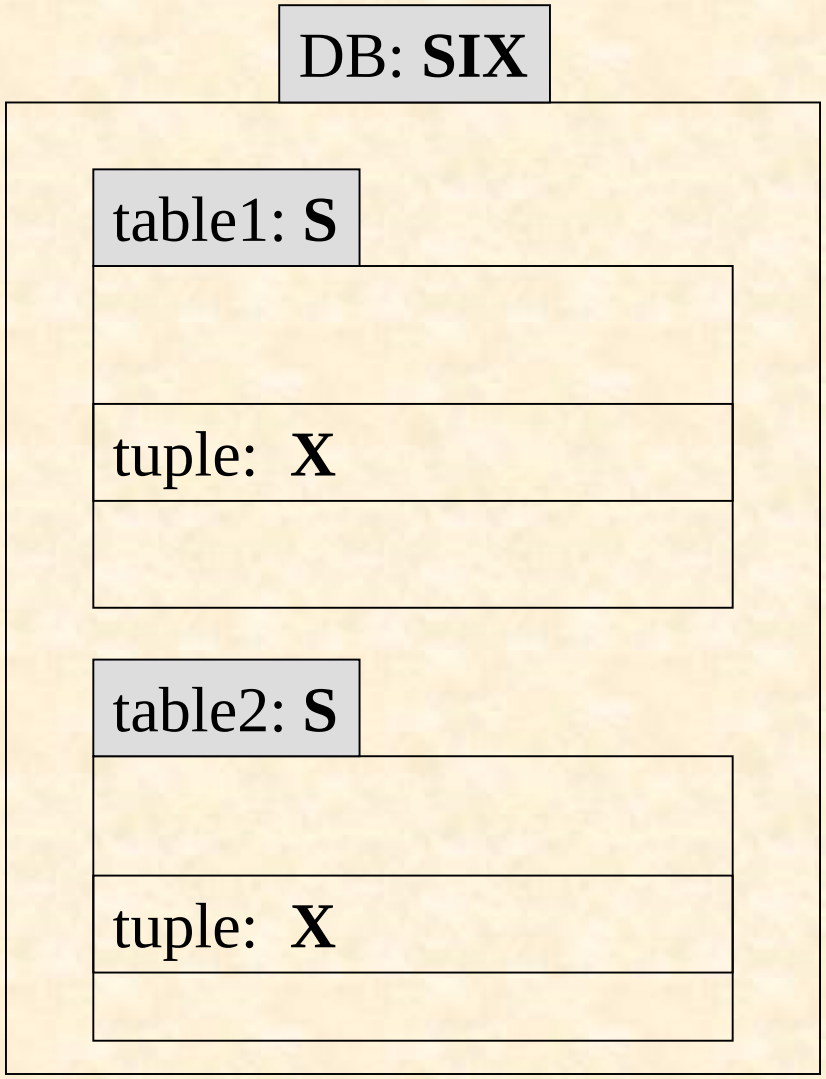

SIX
S
S
X
X

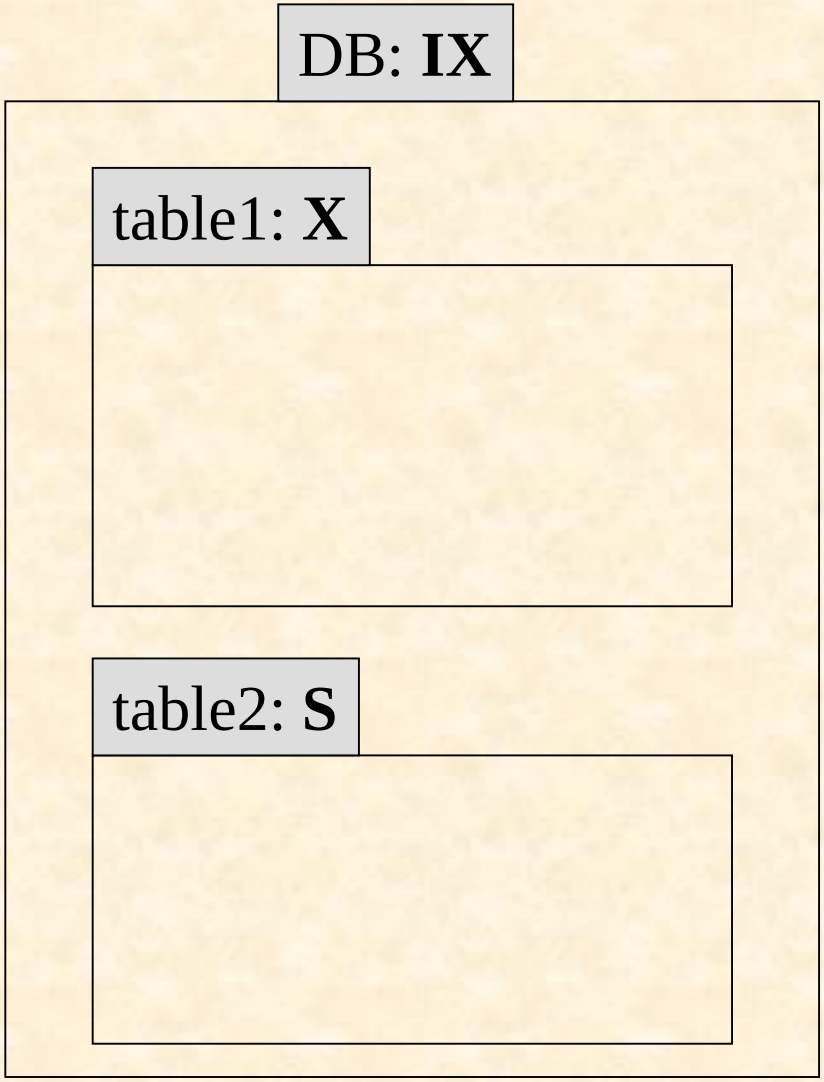

DB: SIX
table1: S
tuple: X
table2: S
tuple: X

```
update  table1
set A=A+1
where A>10
```

```
update  table1
set A=A+1;

select *  from table2
```





Intention Lock Modes (cont.)

S, X : 局部, e.g. tuple
IS, IX, SIX: 整体, e.g. DB, table

- Assuming that T_i requests a lock of *mode **A*** on data item Q on which another transaction T_j currently holds a lock of *mode **B***, if T_i can be granted a lock of *mode A*, then *mode **A** is*
  - refer to Fig. 18.17

Compatibility Matrix with Intention Lock Modes

请求T_i \ 现有T_j	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

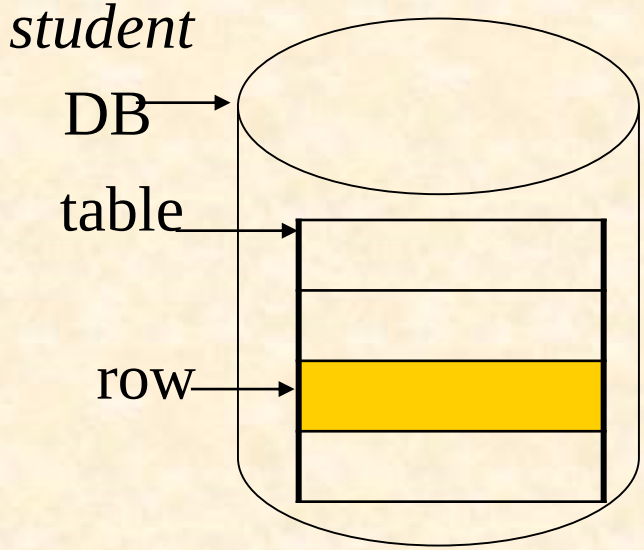
X:
update
insert
bulk insert

Fig. 18.17 Compatibility matrix

E.g.

T_i want to access on the table *student*;

T_j is now accessing on *student*



S, X : 局部

IS, IX, SIX: 整体

IS on *DB*
IS on *table*
S* on *tuple

select *ID*
from *student*
where *age*>0

IS on *DB*

S* on *table

select *ID*
from *student*

IX/SIX on *DB*
IX on *table*
X* on *tuple

update *student*
set *ID*:=*ID*+1
where *age* >20
delete
from *student*
where *age*>0

IX/SIX on *DB*

X* on *table

update *student*
set *ID*:=*ID*+1

delete
from *student*

- 多粒度锁协议要求加锁按自顶向下（根到叶）的顺序，释放锁按照自底向上（叶到根）的顺序
- T_i对数据项Q的加锁规范

  e.g.数据对象层次：DB——table *student*——rows in *student*

  Q为表*student*

1. 遵循Fig.18.17的锁模式相容矩阵
2. 根结点, e.g. DB, 首先加锁, e.g. IS, IX
3. 仅当T_i当前对Q (e.g. student)的父结点(e.g. DB)持有IX、IS锁时， T_i对结点Q可加S或IS型锁，e.g.

—— T_i当前对DB持有IX、IS锁，表明需要对DB中*student*等表写或读

—— T_i当可进一步对*student*加S锁，请求对*student*整体读；或进一步对*student*加IS锁，请求读*student*中某些行读

- 4.仅当T_i当前对Q的父结点持有IX、SIX锁时，T_i对结点Q可加X、IX、SIX型锁，e.g.

—— T_i当前对DB持有IX、SIX锁，表明需要对DB中*student*等表写；

或：需要对DB中*student*等表读，对这些表中的某些行写

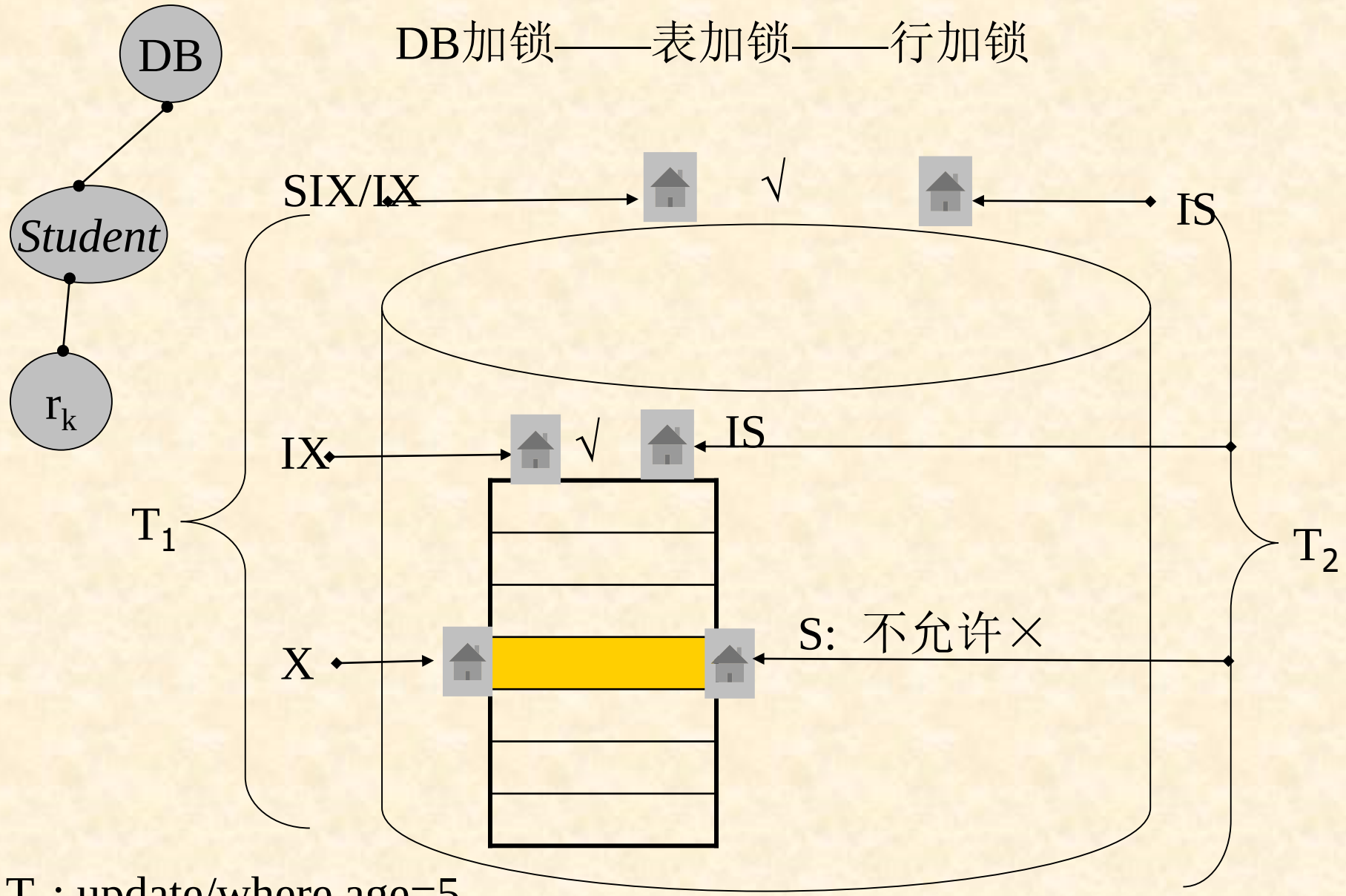
—— T_i当可进一步对*student*加X锁，请求对*student*整体写；

或：进一步对*student*加IX锁，请求读*student*中某些行写；

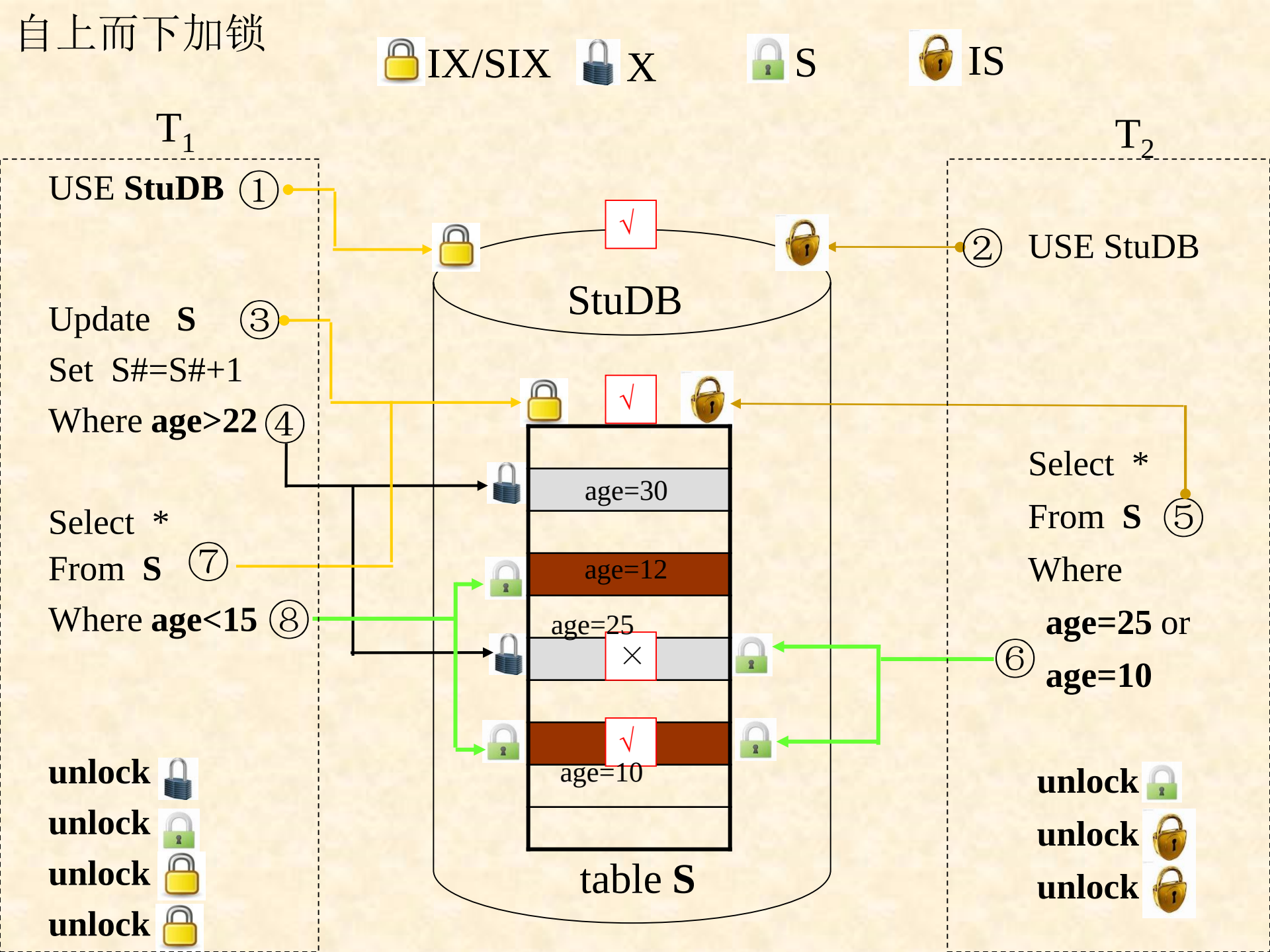
或：进一步对*student*加SIX锁，请求读*student*中某些行读，并对该行中某些属性写;

5. T_i 可以对结点Q加锁， 仅当T_i 以前没有对任何结点解锁，即T_i 满足2PL.

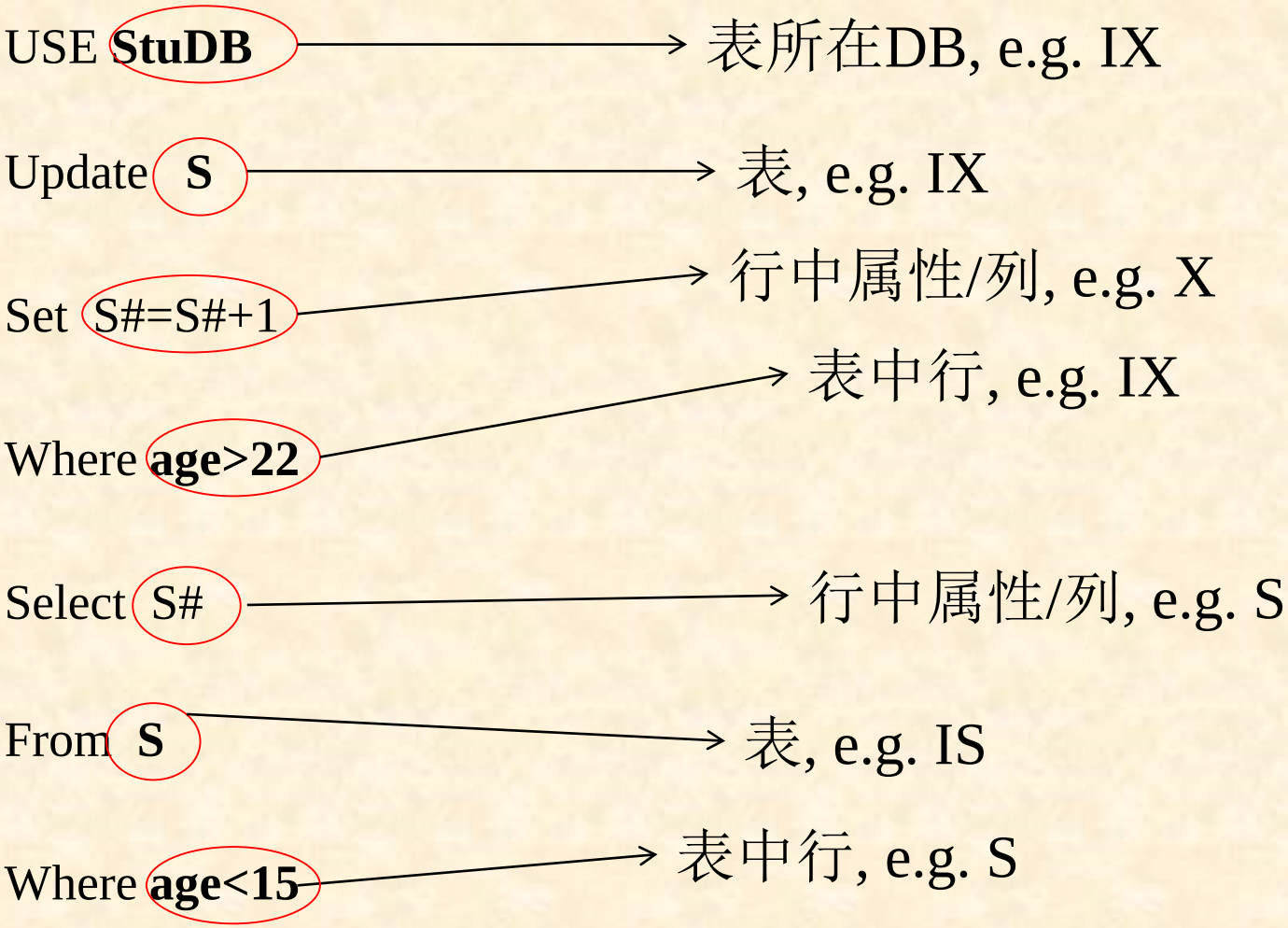
6. T_i可对结点Q解锁，仅当T_i不持有Q的子结点的锁


DB
Student
r_k
DB加锁——表加锁——行加锁
SIX/IX
√
IS
IX
√
IS
T_1
T_2
X
S: 不允许×
T_1: update/where age=5
T_2: select/where age=5

自上而下加锁


IX/SIX
X
S
IS
T1
T2
USE StuDB ①
② USE StuDB
√
StuDB
Update S ③
Set S#=S#+1
Where age>22 ④
√
Select *
From S ⑤
Where
age=25 or
age=10
⑥
age=30
age=12
age=25
×
√
age=10
Select *
From S ⑦
Where age<15 ⑧
table S
unlock
unlock
unlock
unlock
unlock
unlock
unlock

SQL语句涉及到的加锁级别(maybe)



Multiple Granularity Locking Scheme

- Transaction T_i can lock a node Q, using the following rules
  - the lock compatibility matrix in Fig. 18.17 must be observed
  - the root of the tree must be locked first, and may be locked in any mode.
  - a node Q can be locked by T_i in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode
  - a node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.

Multiple Granularity Locking Scheme (cont.)

- T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
- T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order

- Multiple Granularity Locking ensure serializability!!

Homework

1. What is a recoverable schedule? Why is recoverability of schedules desirable?
2. What is a cascadeless schedule? Why is cascadelessness of schedules desirable?
3. What benefit does rigorous two-phase locking provide? How does it compare with other forms of two-phase locking?
4. What benefit does strict two-phase locking provide? What disadvantages result?

Homework

18.2 Consider the following two transactions:

T_{34}: read(A);
read(B);
if $A = 0$ **then** $B := B + 1$;
write(B).

T_{35}: read(B);
read(A);
if $B = 0$ **then** $A := A + 1$;
write(A).

Add lock and unlock instructions to transactions T_{31} and T_{32} so that they observe the two-phase locking protocol. Can the execution of these transactions result in a deadlock?