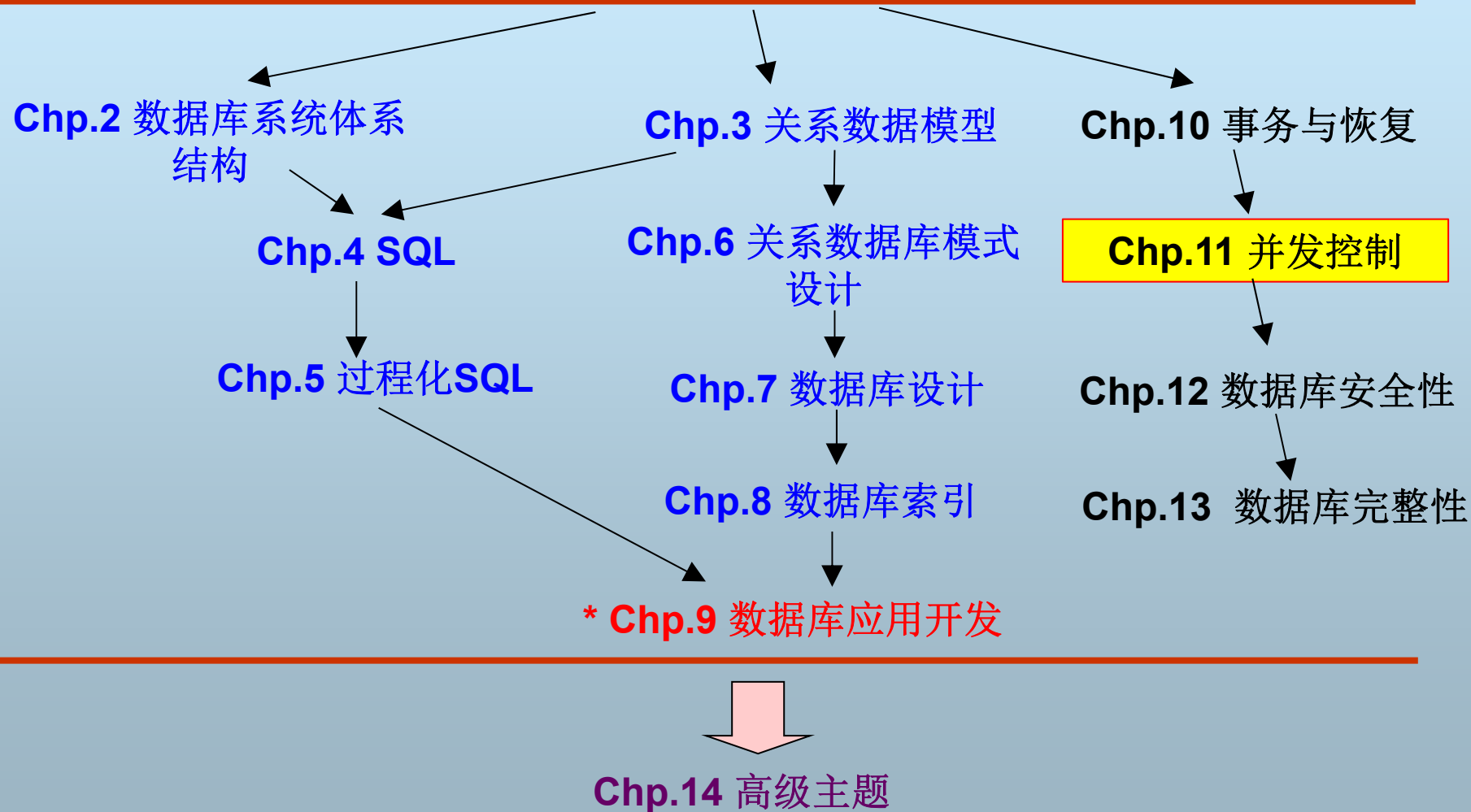


# 第11章 并发控制



# 课程知识结构

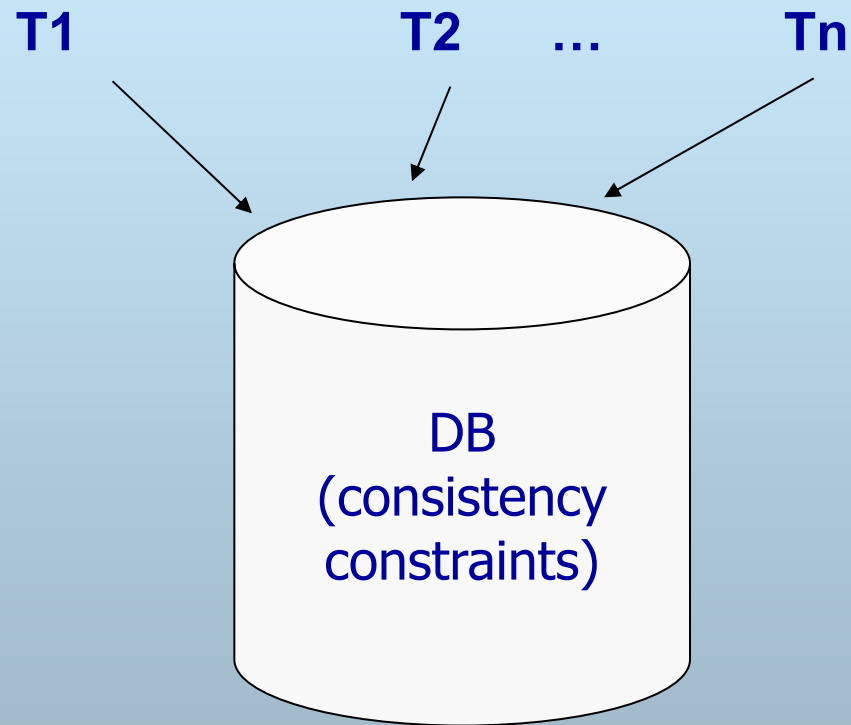
## Chp.1 数据库系统概述



# Databases Protection

- 数据库保护：排除和防止各种对数据库的干扰破坏，确保数据安全可靠，以及在数据库遭到破坏后尽快地恢复
- 数据库保护通过四个方面来实现
  - 完整性控制技术
    - ◆ Enable constraints
  - 安全性控制技术
    - ◆ Authorization and authentication
  - 数据库的恢复技术
    - ◆ Deal with failure
  - 并发控制技术
    - ◆ Deal with data sharing

# Concurrency Control



多个事务同时存取共享的数据库时，如何保证数据库的一致性？

# 主要内容

- 并发操作与并发问题
- 并发事务调度与可串行性  
(Scheduling and Serializability )
- 锁与可串行性实现 (Locks)
- 事务的隔离级别
- 死锁

# 一、并发操作和并发问题

## ■ 并发操作

- 在多用户**DBS**中，如果多个用户同时对同一数据进行操作称为**并发操作**
- 并发操作使多个事务之间可能产生相互干扰，破坏事务的**隔离性**（**Isolation**）
- **DBMS**的并发控制子系统负责协调并发事务的执行，保证数据库的一致性，避免产生不正确的数据

## ■ 并发操作通常会引起三类问题（**三大并发问题**）

- 丢失更新（**Lost update**）
- 脏读（**Dirty read / Uncommitted update**）
- 不一致分析（**Inconsistent analysis**）

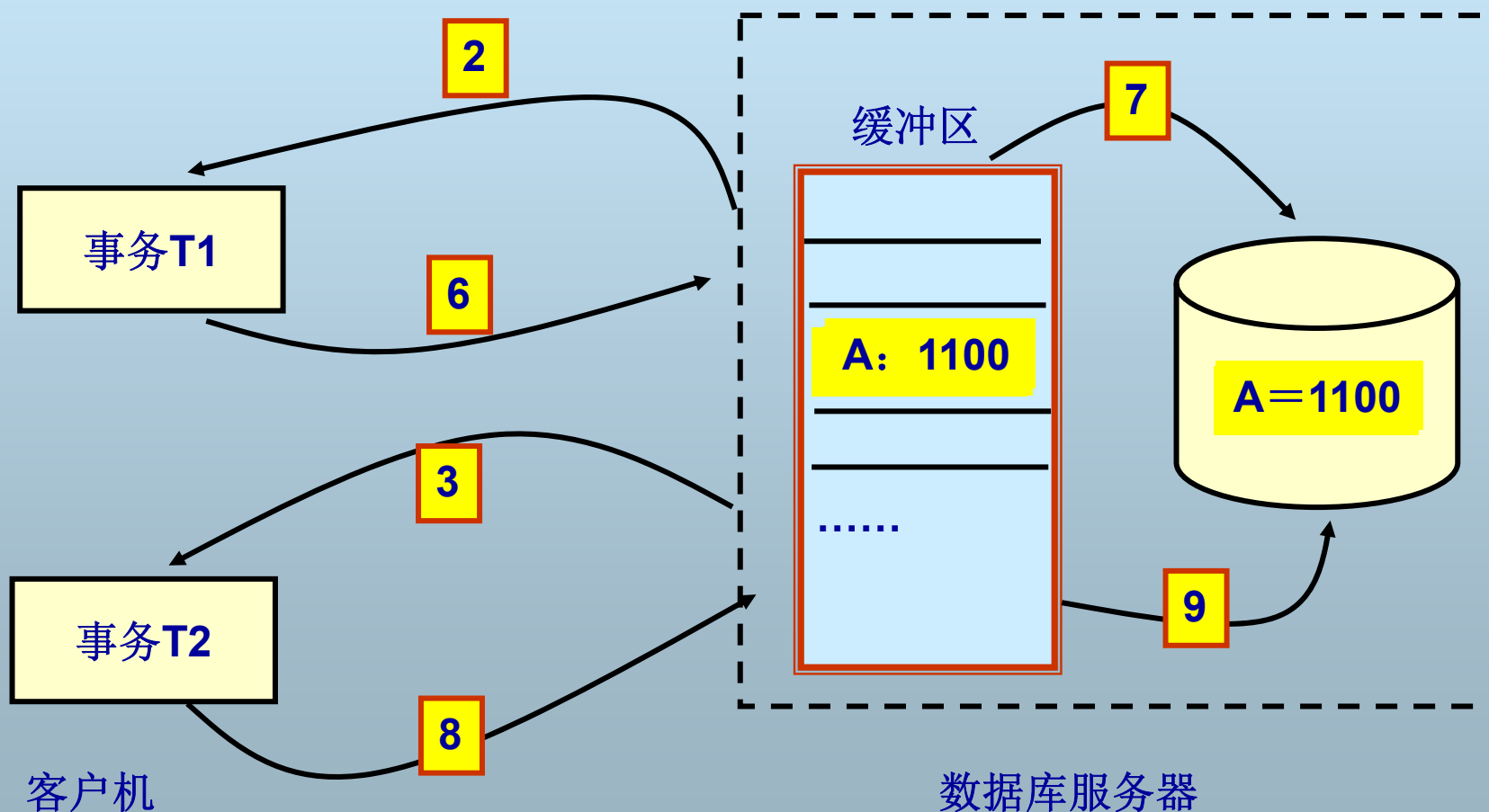
# 1、丢失更新问题

时间	事务T1	事务T2	数据库中A的值
1			1000
2	Read(A,t)		
3		Read(A,t)	
4	t=t-100		
5		t=t+100	
6	Write(A,t)		
7	Commit		900
8		Write(A,t)	
9		Commit	1100

# 基于延迟更新的执行示例

延迟更新：事务Commit后才将所有更新写入数据库

时间	事务T1	事务T2	数据库中A的值
1			1000
2	Read(A,t)		
3		Read(A,t)	
4	t=t-100		
5		t=t+100	
6	Write(A,t)		
7	Commit		900
8		Write(A,t)	
9		Commit	1100





## 2、脏读问题

时间	事务T1	事务T2	数据库中A的值
1			1000
2	Read(A,t)		
3	t=t-100		
4	Write(A,t)		
5		Read(A,t)	
6	Rollback	t=t+100	900
7		Write(A,t)	
8		Commit	1000

**脏数据：事务在内存中更新了但还未最终提交的数据**

### 3、不一致分析问题

时间	事务T1	事务T2
1		
2	Read(A,t)	Read(B,t)
3	t=t-100	
4		Read(A,v)
5	Write(A,t)	
6	Commit	
7		Sum=t+v
8		Commit

不一致分析问题：事务读了过时的数据

# 并发控制问题讨论

■ 下图所示的并发调度存在什么问题？

A. 丢失更新

C. 脏读

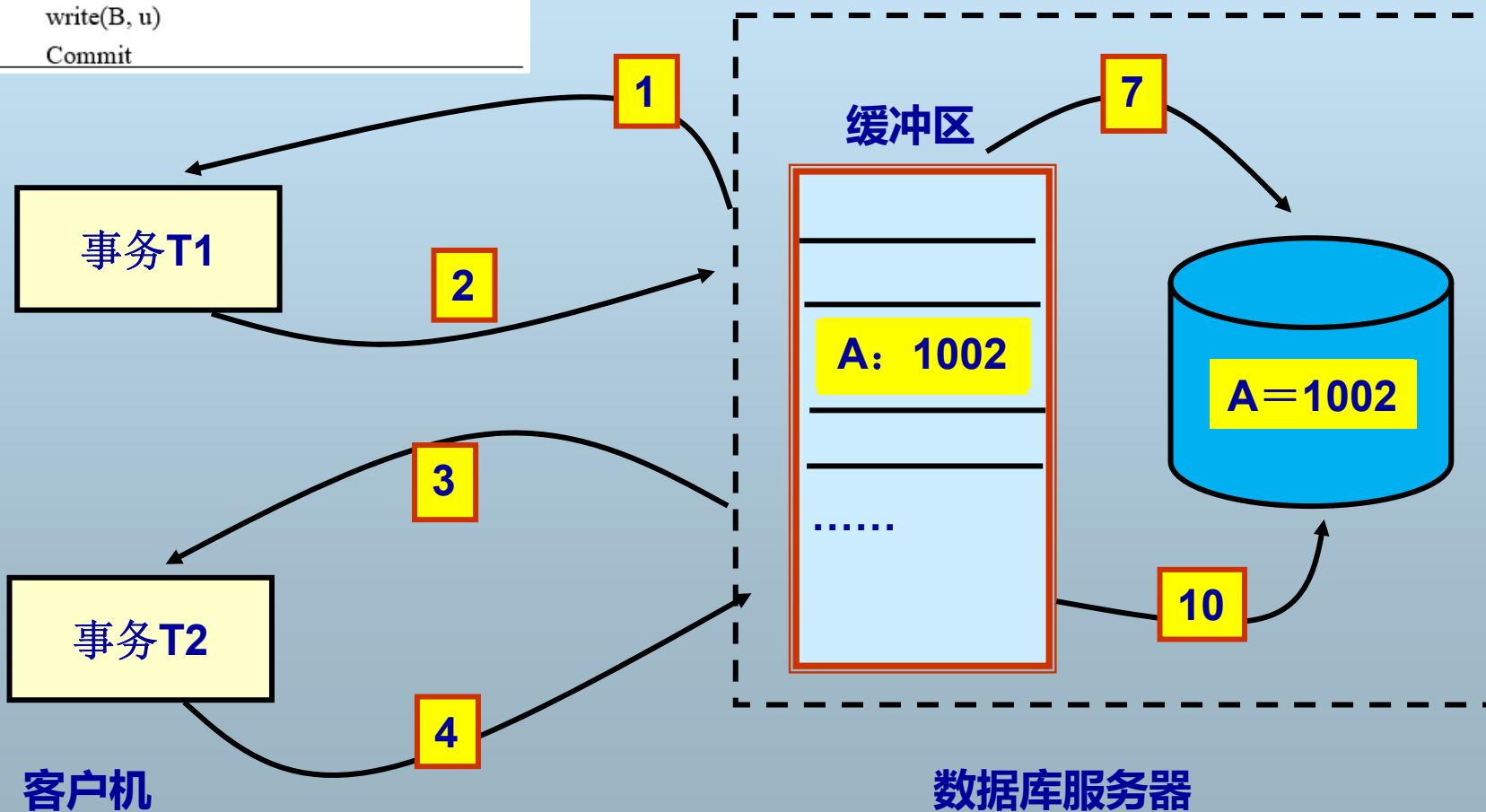
B. 丢失更新、脏读

D. 丢失更新、脏读、不一致分析

Step	T1	T2
1	read(A, t), t=t+1	
2	write(A,t)	
3		read(A, t), t = t+1
4		write(A, t)
5		read(B, u), u = u+1
6		write(B, u)
7		Commit
8	read(B, u), u =u+1	
9	write(B, u)	
10	Commit	

Step	T1	T2
1	read(A, t), t=t+1	
2	write(A,t)	
3		read(A, t), t = t+1
4		write(A, t)
5		read(B, u), u = u+1
6		write(B, u)
7		Commit
8	read(B, u), u =u+1	
9	write(B, u)	
10	Commit	

## 执行过程分析



# 再论丢失更新问题

Step	T1	T2
1	read(A, t), t = t+1	
2		read(A, t), t = t+1
3	write(A, t)	
3		write(A, t)
4		Commit
6	Commit	

## Lost update

两次提交写导致的写覆盖

Step	T1	T2
1	read(A, t), t = t+1	
2		read(A, t), t = t+1
3	write(A, t)	
4		write(A, t)
5		read(B, u), u = u+1
6		write(B, u)
7		Commit
8	Abort	

## Dirty write

由于Rollback导致的提交事务的写失效  
破坏了T2的原子性

DBMS中不允许出现Dirty write  
在任何情况下都要求X锁保留到事务结束



[Hal Berenson](#), [Philip A. Bernstein](#), [Jim Gray](#), [Jim Melton](#), [Elizabeth J. O'Neil](#), [Patrick E. O'Neil](#):  
A Critique of ANSI SQL Isolation Levels. [SIGMOD 1995](#): 1-10

# 4、并发控制的问题该如何解决？

## ■ 一种方法

### ● 让所有事务一个一个地串行执行

- ◆ 一个事务在执行时其它事务只能等待
- ◆ 不能充分利用系统资源，效率低下

## ■ 另一种方法

### ● 为了充分发挥**DBMS**共享数据的特点，应允许事务内部的读写操作并发执行

### ● 挑战

- ◆ 必须保证事务并发执行的正确性；必须用正确的方法调度执行事务的并发操作

## 二、调度(Schedule)

### Example

T1:    Read(A, t)  
       $t \leftarrow t+100$   
      Write(A, t)  
      Read(B, t)  
       $t \leftarrow t+100$   
      Write(B, t)

T2:    Read(A, s)  
       $s \leftarrow s \times 2$   
      Write(A, s)  
      Read(B, s)  
       $s \leftarrow s \times 2$   
      Write(B, s)

Constraint:  $A=B$

## 二、调度(Schedule)

### Schedule A

T1	T2	A	B
Read(A, t); $t \leftarrow t+100$		25	25
Write(A, t);		125	25
Read(B, t); $t \leftarrow t+100$ ;			
Write(B, t);		125	125
	Read(A, s); $s \leftarrow s \times 2$ ;		
	Write(A, s);	250	125
	Read(B, s); $s \leftarrow s \times 2$ ;		
	Write(B, s);	250	250



## 二、调度(Schedule)

### Schedule B

T1	T2	A	B
	Read(A, s); $s \leftarrow s \times 2$ ;	25	25
	Write(A, s);	50	25
	Read(B, s); $s \leftarrow s \times 2$ ;		
	Write(B, s);	50	50
Read(A, t); $t \leftarrow t + 100$			
Write(A, t);		150	50
Read(B, t); $t \leftarrow t + 100$ ;			
Write(B, t);		150	150

## 二、调度(Schedule)

### Schedule C

T1	T2	A	B
Read(A, t); $t \leftarrow t+100$		25	25
Write(A, t);		125	25
	Read(A, s); $s \leftarrow s \times 2$ ;		
	Write(A, s);	250	25
Read(B, t);			
$t \leftarrow t+100$ ;			
Write(B, t);		250	125
	Read(B, s); $s \leftarrow s \times 2$ ;		
	Write(B, s);	250	250

## 二、调度(Schedule)

### Schedule D

T1	T2	A	B
Read(A, t); $t \leftarrow t+100$		25	25
Write(A, t);		125	25
	Read(A, s); $s \leftarrow s \times 2$ ;		
	Write(A, s);	250	25
	Read(B, s); $s \leftarrow s \times 2$ ;		
	Write(B, s);	250	50
Read(B, t);			
$t \leftarrow t+100$ ;			
Write(B, t);		250	150

## 二、调度(Schedule)

### Schedule D

T1	T2'	A	B
Read(A, t); $t \leftarrow t+100$		25	25
Write(A, t);		125	25
	Read(A, s); $s \leftarrow s \times 1$ ;		
	Write(A, s);	125	25
	Read(B, s); $s \leftarrow s \times 1$ ;		
	Write(B, s);	125	25
Read(B, t);			
$t \leftarrow t+100$ ;			
Write(B, t);		125	125

# 1、调度的定义

## ■ 调度

- 多个事务的读写操作按时间排序的执行序列

T1: r1(A) w1(A) r1(B) w1(B)

T2: r2(A) w2(A) r2(B) w2(B)

Sc = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)

## Note

- 调度中每个事务的读写操作保持原来顺序
- 事务调度时不考虑
  - ◆ 数据库的初始状态 (Initial state)
  - ◆ 事务的语义 (Transaction semantics)

# 1、调度的定义

## ■ 多个事务的并发执行存在多种调度方式

Example:

$Sc = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)$

$Sa = r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)$

T1

T2

What is a correct schedule?

And how to get a correct schedule?

## 2、可串化调度 (Serializable Schedule)

### ■ What is a correct schedule?

- Answer: a serializable schedule!

### ■ 串行调度 (Serial schedule)

- 各个事务之间没有任何操作交错执行，事务一个  
一个执行
- $S = T1 T2 T3 \dots Tn$

### ■ Serializable Schedule

- 如果一个调度的结果与某一串行调度执行的结果等价，则称该调度是可串化调度，否则是不可串调度

## 2、可串化调度

### ■ 可串化调度的正确性

- **Consistence of transaction:** 单个事务的执行保证**DB**从一个一致状态变化到另一个一致状态
- **N个事务串行调度执行仍保证 Consistence of DB**





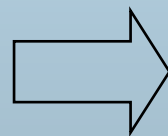
## 2、可串行化调度

### ■ Is a schedule a serializable one?

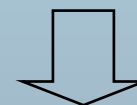
#### ● We MUST

- ◆ Get all results of serial schedules,  **$n!$**
- ◆ See if the schedule is equivalent to some serial schedule

Too difficult to  
realize



Other approaches?



冲突可串行性

### 3、冲突可串性 (conflict serializable)

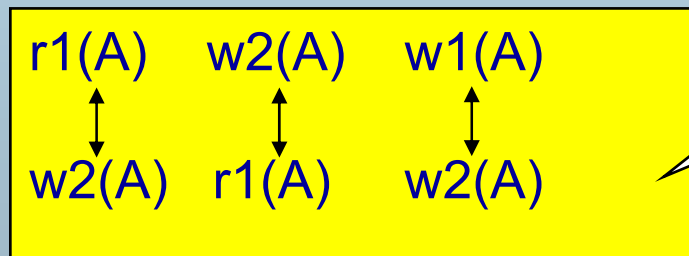
#### ■ Conflicting actions

##### ● Say

◆  $r_i(X)$  : 事务  $T_i$  的读  $X$  操作 ( $Read(X, t)$ )

◆  $w_i(X)$  : 事务  $T_i$  的写  $X$  操作 ( $Write(X, t)$ )

##### ● 冲突操作



涉及同一个数据库元素，  
并且至少有一个是写操作

### 3、冲突可串性 (conflict serializable)

#### ■ Conflicting actions

- 如果调度中一对连续操作是冲突的，则意味着如果它们的执行顺序交换，则至少会改变其中一个事务的最终执行结果
- 如果两个连续操作不冲突，则可以在调度中交换顺序

### 3、冲突可串性

#### Schedule C

T1	T2	A	B
Read(A, t); $t \leftarrow t+100$		25	25
Write(A, t);		125	25
	Read(A, s); $s \leftarrow s \times 2$ ;		
	Write(A, s);	250	25
Read(B, t);			
$t \leftarrow t+100$ ;			
Write(B, t);		250	125
	Read(B, s); $s \leftarrow s \times 2$ ;		
	Write(B, s);	250	250

$Sc = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)$

### 3、冲突可串行性

$Sc = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)$

$Sc' = r1(A) w1(A) r2(A) r1(B) w2(A) w1(B) r2(B) w2(B)$

$Sc' = r1(A) w1(A) r1(B) r2(A) w2(A) w1(B) r2(B) w2(B)$

$Sc' = r1(A) w1(A) r1(B) r2(A) w1(B) w2(A) r2(B) w2(B)$

$Sc' = r1(A) w1(A) r1(B) w1(B) r2(A) w2(A) r2(B) w2(B)$

T1

T2

Schedule A

### 3、冲突可串行性

$Sc = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)$

同一个事务的操作必须符合原来的顺序

冲突操作

### 3、冲突可串性

Schedule C

此步读入的B为25

T1	T2	A	B
Read(A, t); $t \leftarrow t+100$		25	25
Write(A, t);		125	25
	Read(A, s); $s \leftarrow s \times 2$ ;		
	Write(A, s);	250	25
Read(B, t);			
$t \leftarrow t+100$ ;			
	Read(B, s); $s \leftarrow s \times 2$ ;		
Write(B, t);		250	125
	Write(B, s);	250	50

### 3、冲突可串性

- **冲突等价 (conflict equivalent )**
  - **S1, S2 are conflict equivalent schedules if S1 can be transformed into S2 by a series of swaps on non-conflicting actions.**
- **冲突可串性 (conflict serializable)**
  - **A schedule is conflict serializable if it is conflict equivalent to some serial schedule.**



# 3、冲突可串行性

## ■ 定理

- 如果一个调度满足冲突可串行性，则该调度是可串行化调度

## ■ Note

- 仅为充分条件

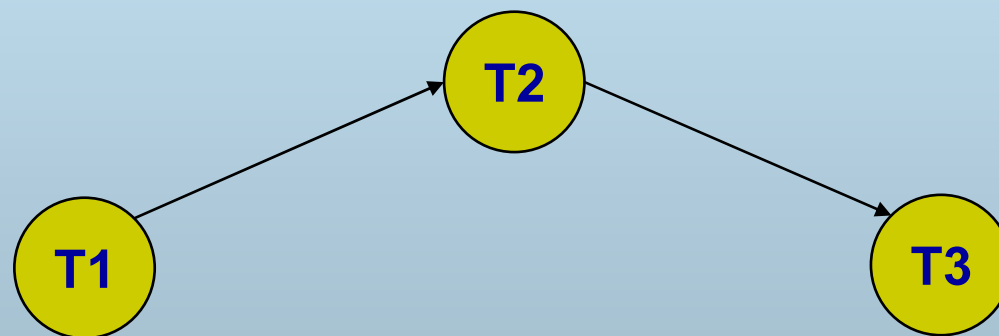
## 4、优先图 (Precedence Graph)

- 优先图用于冲突可串性的判断
- 优先图结构
  - 结点 (Node): 事务
  - 有向边 (Arc):  $T_i \rightarrow T_j$  , 满足  $T_i <_s T_j$ 
    - ◆ 存在 $T_i$ 中的操作A1和 $T_j$ 中的操作A2, 满足
      - A1在A2前, 并且
      - A1和A2是冲突操作

## 4、优先图

### Example

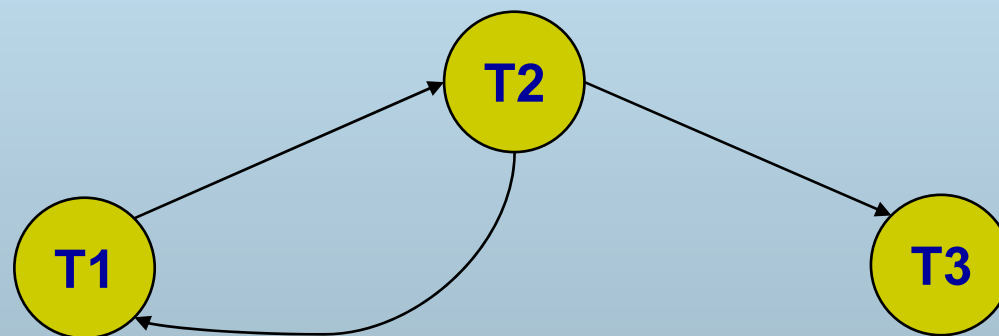
$S = r_2(A) \ r_1(B) \ w_2(A) \ r_3(A) \ w_1(B) \ w_3(A) \ r_2(B) \ w_2(B)$



## 4、优先图

### Example

$S = r_2(A) \ r_1(B) \ w_2(A) \ r_2(B) \ r_3(A) \ w_1(B) \ w_3(A) \ w_2(B)$




## 4、优先图

### ■ 优先图与冲突可串行性

- 给定一个调度 $S$ ，构造 $S$ 的优先图 $P(S)$ ，若 $P(S)$ 中无环，则 $S$ 满足冲突可串行性
- 证明：归纳法
  - ◆ see “H. Molina et al. *Database System Implementation*”

# Next

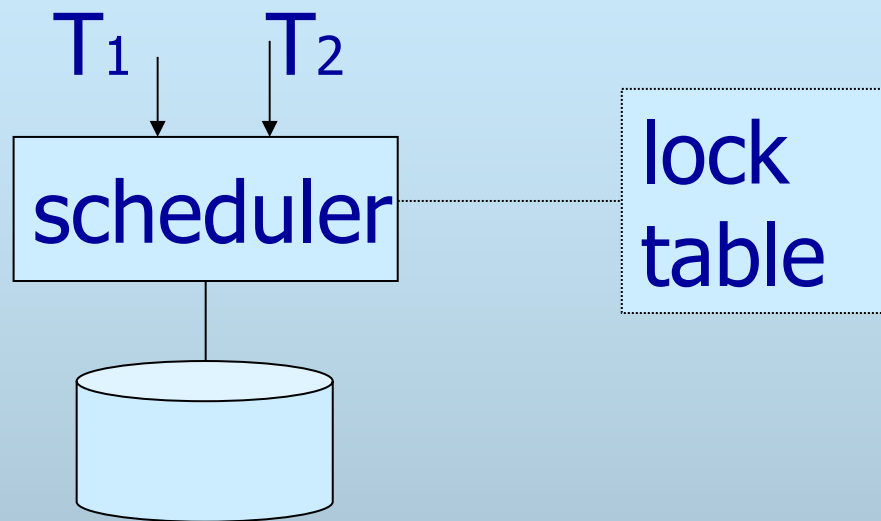
- 并发操作与并发问题
- 并发事务调度与可串行性  
(Scheduling and Serializability )
- 锁与可串行性实现 (Locks) 
- 事务的隔离级别

## 三、锁与可串行性实现

- **What is a correct schedule?**
  - **a serializable schedule!**
- **How to get a serializable schedule?**
  - **Using locks**

**给定n个并发事务，确定一个可串行化调度**

# 1、锁简介



Two new actions:

lock (exclusive):  $l_i(A)$

unlock:  $u_i(A)$




## 1、锁简介

## ■ 锁协议(protocol): 使用锁的规则

## Rule #1: Well-formed transactions

**Ti: ... li(A) ... pi(A) ... ui(A) ...**

## Rule #2 Legal scheduler

**S** = ..... li(A) ..... ui(A) .....  
  
 no lj(A)

# 1、锁简介

$S = r2(A) \ r1(B) \ w2(A) \ r2(B) \ w1(B) \ w2(B)$

$S = I2(A) \ r2(A) \ I1(B) \ r1(B) \ w2(A) \ u2(A) \ I2(B) \ r2(B) \ w1(B) \ u1(B) \ w2(B) \ u2(B)$

**Well-formed but  
illegal**

## 2、两阶段锁(2PL)

### ■ Two Phase Locking

$T_i = \dots\dots li(A) \dots\dots ui(A) \dots\dots$

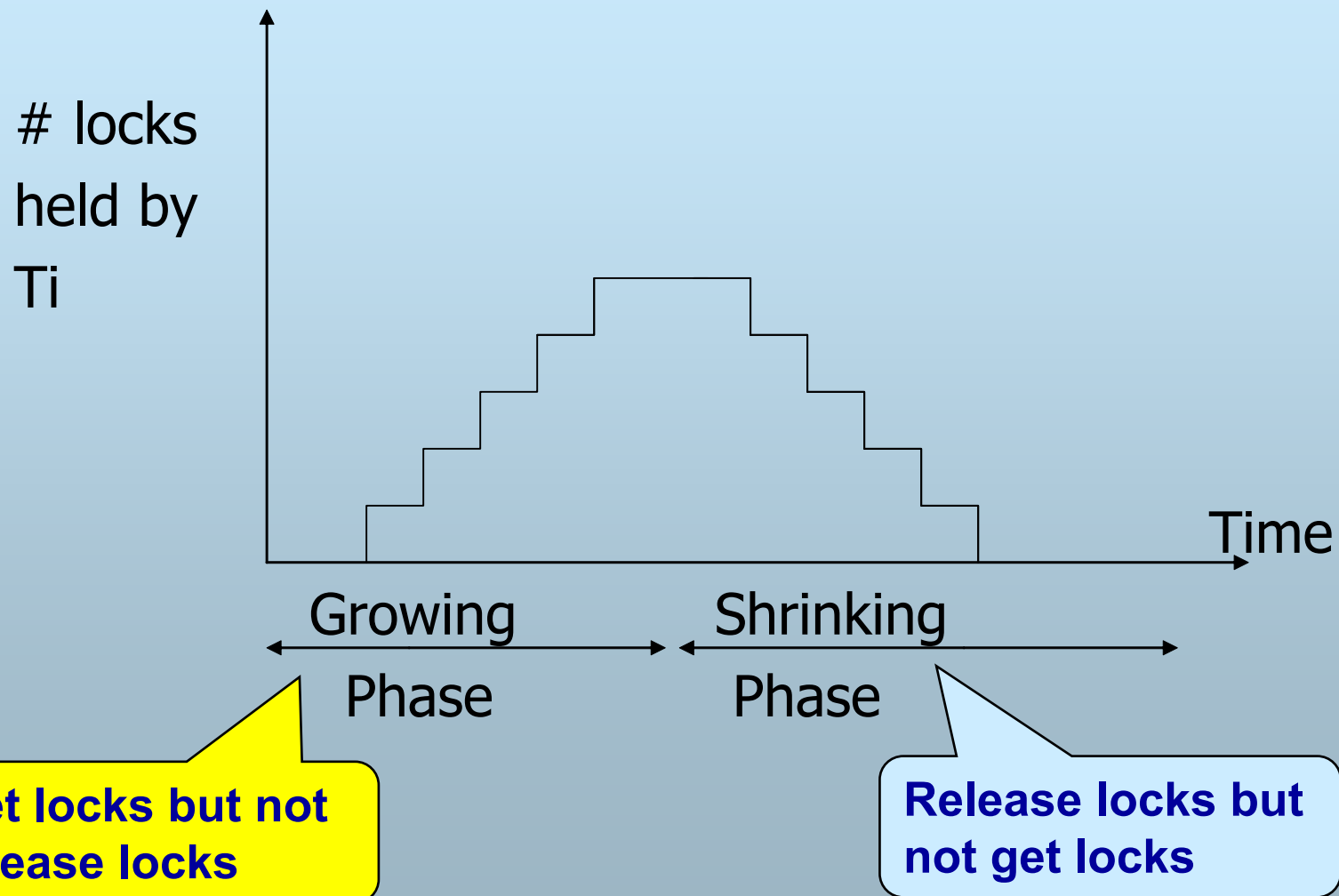


1. 事务在对任何数据进行读写之前，首先要获得该数据上的锁
2. 在释放一个锁之后，事务不再获得任何锁

Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, Irving L. Traiger:

The Notions of Consistency and Predicate Locks in a Database System. Commun. ACM 19(11): 624-633 (1976)

## 2、两阶段锁(2PL)

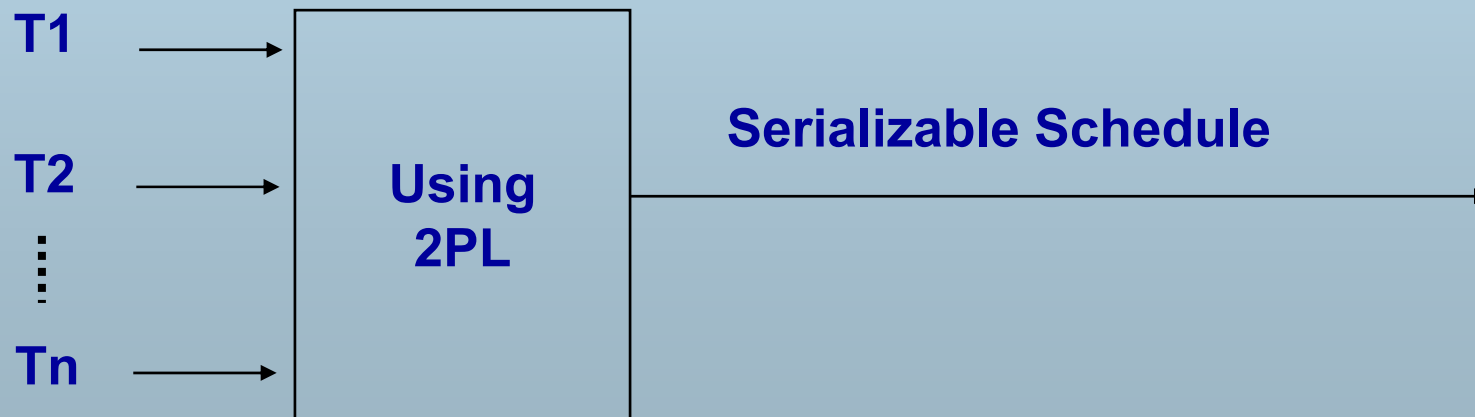


## 2、两阶段锁(2PL)

- 两段式事务：遵守2PL协议的事务

- 定理

- 如果一个调度S中的所有事务都是两段式事务，则该调度是可串化调度



## 2、两阶段锁(2PL)

- 如果事务T只是读取X，也必须加锁，而且释放锁之前其它事务无法对X操作，影响数据库的并发性
- 解决方法
  - 引入不同的锁，满足不同的要求
    - ◆ S Lock
    - ◆ X Lock
    - ◆ Update Lock
    - ◆ .....

### 3、X Lock

- **Exclusive Locks (X锁, 也称写锁)**
- **X锁**: 若事务T对数据R加X锁, 那么其它事务要等T释放X锁以后, 才能获准对数据R进行封锁。只有获得R上的X锁的事务, 才能对所封锁的数据进行 **修改**。

# 3、X Lock

## Example

## Using X lock for schedules

T1	T2	A	B
Read(A, t); $t \leftarrow t+100$		25	25
Write(A, t);		125	25
	Read(A, s); $s \leftarrow s \times 2$ ;		
	Write(A, s);	250	25
Read(B, t);			
$t \leftarrow t+100$ ;			
	Read(B, s); $s \leftarrow s \times 2$ ;		
Write(B, t);		250	125
	Write(B, s);	250	50

An incorrect  
schedule



# 3、X Lock

T1	T2	A	B
<b>xL1(A)</b>		25	25
Read(A, t); $t \leftarrow t+100$			
Write(A, t);		125	25
<b>xL1(B)</b>	<b>xL2(A)</b>		
Read(B, t); $t \leftarrow t+100$ ;	wait		
Write(B, t);	wait		
<b>U1(A)</b>	wait	125	125
<b>U1(B)</b>	Read(A, s); $s \leftarrow s \times 2$ ;		
	Write(A, t)	250	125
	<b>xL2(B)</b>		
	Read(B, s); $s \leftarrow s \times 2$ ;		
	Write(B, s);	250	250
	<b>U2(A)</b>		
	<b>U2(B)</b>		

**X-lock-based 2PL**

## 3、X Lock

- X锁提供了对事务的写操作的正确控制策略
- 但如果事务是只读事务，则没必要加X锁
  - 写——独占
  - 读——共享

## 4、S Lock

- **Share Locks (S锁, 也称读锁)**
- **S锁**: 如果事务T对数据R加了S锁, 则其它事务对R的X锁请求不能成功, 但对R的S锁请求可以成功。这就保证了其它事务可以读取R但不能修改R, 直到事务T释放S锁。当事务获得S锁后, 如果要对数据R进行修改, 则必须在修改前执行Upgrade(R)操作, 将S锁升级为X锁。

## 4、S Lock

### S/X-lock-based 2PL

1. 事务在读取数据R前必须先获得S锁
2. 事务在更新数据R前必须要获得X锁。如果该事务已具有R上的S锁，则必须将S锁升级为X锁
3. 如果事务对锁的请求因为与其它事务已具有的**锁不相容**而被拒绝，则事务进入等待状态，直到其它事务释放锁。
4. 一旦释放一个锁，就不再请求任何锁

## 5、 Compatibility of locks

Requests				
Holds	T1 \ T2	X锁	S锁	无
	X锁	N	N	Y
	S锁	N	Y	Y
	无	Y	Y	Y

- **N: No**, 不相容的请求
- **Y: Yes**, 相容的请求
- 如果两个锁不相容, 则后提出锁请求的事务必须等待

# 6、Update Lock

## Example

t	T1	T2
1	sL1(A)	
2		sL2(A)
3	Read(A)	Read(A)
4		A=A+100
5		Upgrade(A)
6	A=A+100	Wait
7	Upgrade(A)	Wait
8	Wait	Wait
9	Wait	Wait
10	.....	.....

# 6、Update Lock

## ■ Update Lock

- 如果事务取得了数据R上的更新锁，则可以读R，并且可以在以后升级为X锁
- 单纯的S锁不能升级为X锁
- 如果事务持有了R上的Update Lock，则其它事务不能得到R上的S锁、X锁以及Update锁
- 如果事务持有了R上的S Lock，则其它事务可以获取R上的Update Lock

## 6、Update Lock

### ■ 相容性矩阵

	S	X	U
S	Y	N	Y
X	N	N	N
U	N	N	N

#### NOTE

**<S, U>是相容的**：如果其它事务已经持有了S锁，则当前事务可以请求U锁，以获得较好的并发性

**<U, S>不相容**：如果某个事务已持有U锁，则其它事务不能再获得S锁，因为持有U锁的事务可能会由于新的S锁而导致永远没有机会升级到X锁



# 6、Update Lock

## Example

t	T1	T2
1	uL1(A)	
2		uL2(A)
3	Read(A)	Wait
4		Wait
5		wait
6	A=A+100	Wait
7	Upgrade(A)	Wait
8	Write(A)	Wait
9	U1(A)	Wait
10		Read(A)
11		.....

# Where are we ?

- 并发操作与并发问题
- 并发调度与可串行性
- 锁与可串行性实现

- **2PL**

- ◆ S Lock

- ◆ X Lock

- ◆ U Lock

- **Multi-granularity Lock 多粒度锁**

- **Intension Lock 意向锁**



# 8、Multi-Granularity Lock

## ■ Lock Granularity

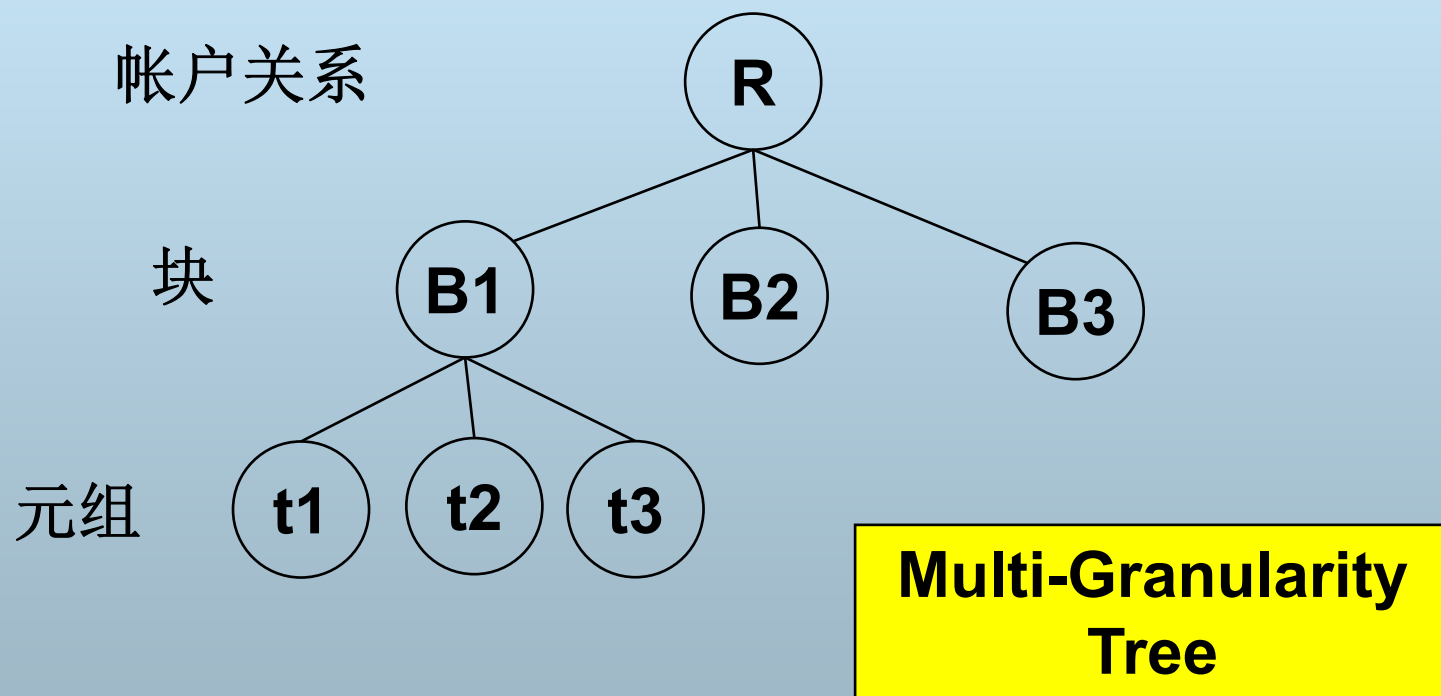
- 指加锁的数据对象的大小

- ◆ 可以是整个关系、块、元组、整个索引、索引项

## ■ 锁粒度越细，并发度越高；锁粒度越粗，并发度越低

## 8、Multi-Granularity Lock

- 多粒度锁：同时支持多种不同的锁粒度



# 8、Multi-Granularity Lock

## ■ 多粒度锁协议

- 允许多粒度树中的每个结点被独立地加**S**锁或**X**锁，对某个结点加锁意味着其下层结点也被加了同类型的锁

# 8、Multi-Granularity Lock

## ■ Why we need MGL?

## 8、Multi-Granularity Lock

**T1:** 求当前数据库中所有帐户的余额之和

**T2:** 增加一个新帐户(余额为1000)

Use tuple locks, suppose total two tuples in R

T1	T2
sL1(o1)	
sL1(o2)	
Read(o1,t1)	
Read(o2,t2)	
<b>Sum=t1+t2</b>	Write(o3)
U1(o1)	
U1(o2)	

并不是当前数据库的实际状态

# 8、Multi-Granularity Lock

## ■ 原因

- **Lock**只能针对已存在的元组，对于开始时不存在后来被插入的元组无法**Lock**
- **o3: Phantom tuple 幻像元组**
  - ◆ 存在，却看不到物理实体

## Solution

- **T2**插入**o3**的操作看成是整个关系的写操作，对整个关系加锁
  - ◆ **Need MGL !**



# 8、Multi-Granularity Lock

**Solution: Using MGL**

<b>T1</b>	<b>T2</b>
<b>sL1(o1)</b>	
<b>sL1(o2)</b>	
<b>Read(o1,t1)</b>	
<b>Read(o2,t2)</b>	
	<b>xL2(R)</b>
<b>Sum=t1+t2</b>	<b>wait</b>
<b>U1(o1)</b>	<b>wait</b>
<b>U1(o2)</b>	<b>wait</b>
	<b>write(o3)</b>

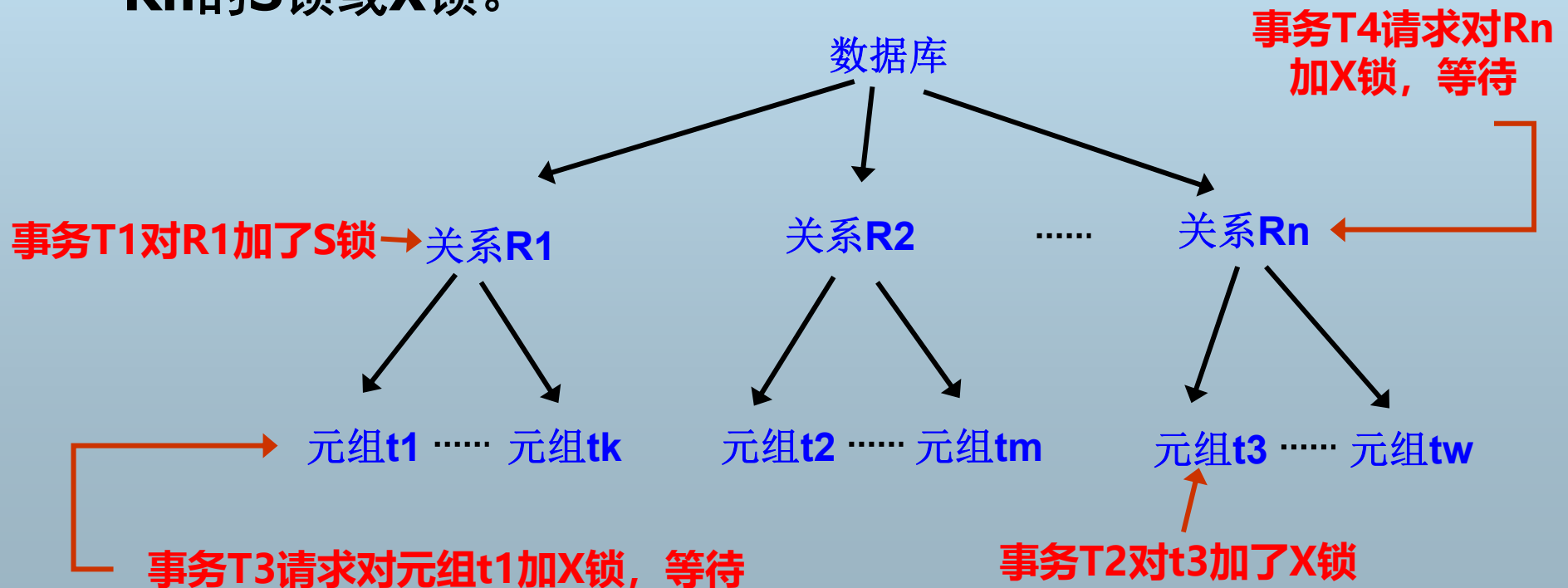
# 8、Multi-Granularity Lock

## ■ 多粒度锁上的两种不同加锁方式

- **显式加锁**：应事务的请求直接加到数据对象上的锁
- **隐式加锁**：本身没有被显式加锁，但因为其上层结点加了锁而使数据对象被加锁
- 给一个结点显式加锁时必须考虑
  - ◆ 该结点是否已有不相容锁存在
  - ◆ 上层结点是否已有不相容的锁（上层结点导致的隐式锁冲突）
  - ◆ 所有下层结点中是否存在不相容的显式锁

## 8、Multi-Granularity Lock

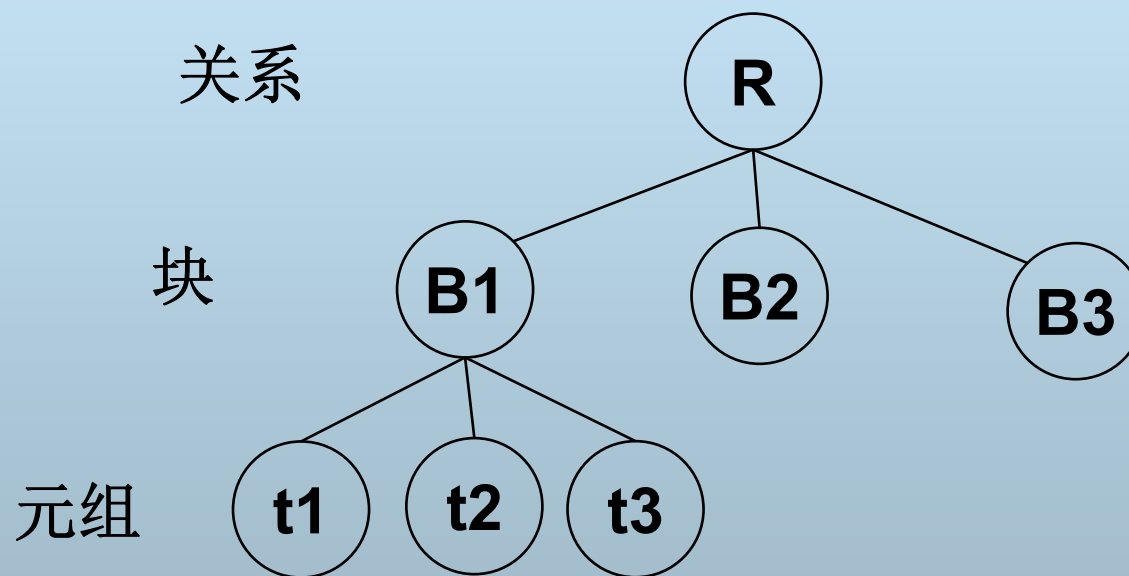
- 事务T1对关系R1显式加了S锁，意味着R1的所有元组也被隐式加了S锁。其它事务可以在R1的元组上加S锁，但不能加X锁
- 事务T2对元组t3加了X锁，其它事务不能请求对其上层结点Rn的S锁或X锁。



## 8、Multi-Granularity Lock

- 在对一个结点P请求锁时，必须判断该结点上是否存在不相容的锁
  - 有可能是P上的显式锁
  - 也有可能是P的上层结点导致的隐式锁
  - 还有可能是P的下层结点中已存在的某个显式锁
- 理论上要搜索上面全部的可能情况，才能确定P上的锁请求能否成功
  - 显然是低效的
  - 引入意向锁 (Intension Lock) 解决这一问题

# 9、Intension Lock



## 9、Intension Lock

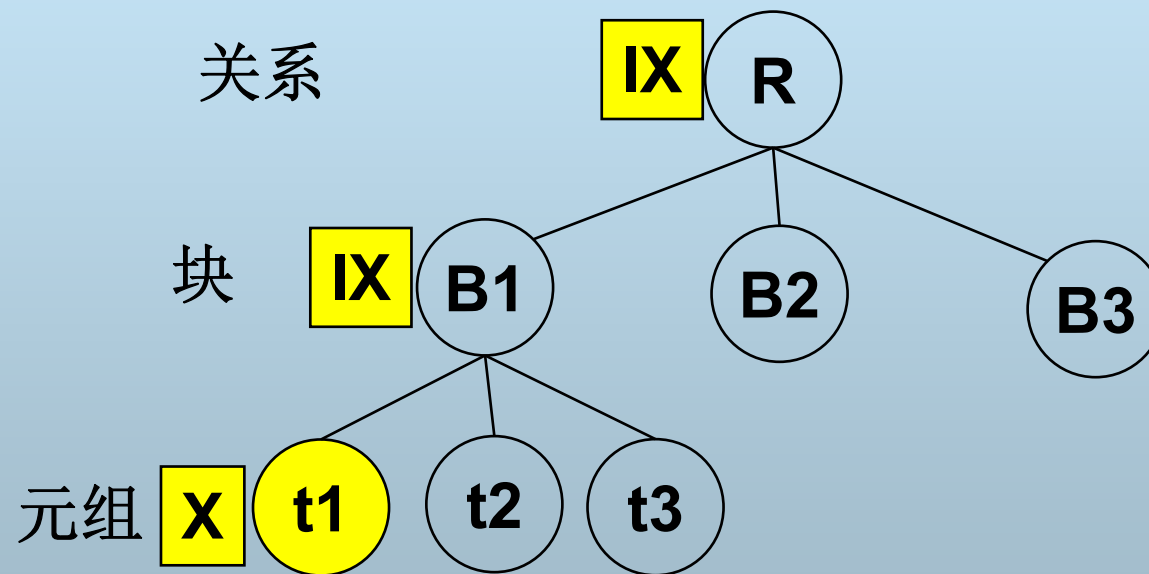
- **IS锁（Intent Share Lock，意向共享锁，意向读锁）**
- **IX锁（Intent Exclusive Lock，意向排它锁，意向写锁）**

## 9、Intension Lock

- 如果对某个结点加**IS(IX)**锁，则说明事务要对该结点的某个下层结点加**S (X)**锁；
- 对任一结点**P**加**S(X)**锁，必须先对从根结点到**P**的路径上的所有结点加**IS(IX)**锁

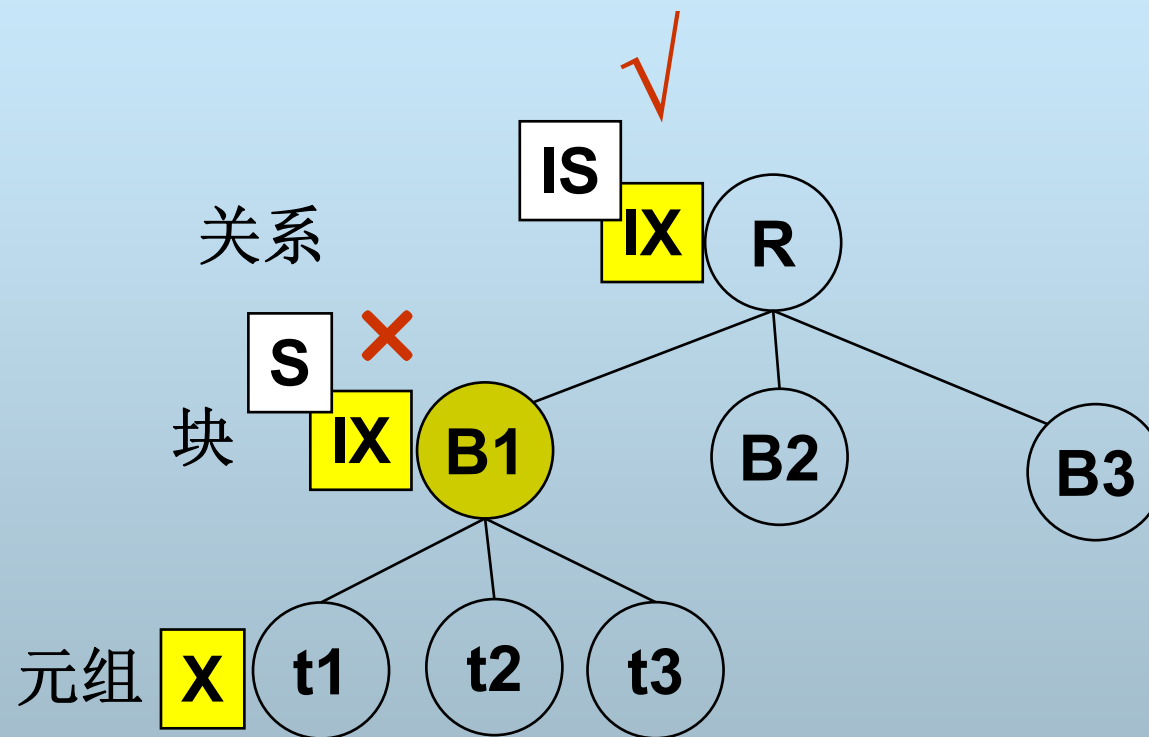
# 9、Intension Lock

Want to exclusively lock t1





## 9、Intension Lock



# 9、Intension Lock

## ■ Compatibility Matrix

	IS	IX	S	X
IS	✓	✓	✓	×
IX	✓	✓	×	×
S	✓	×	✓	×
X	×	×	×	×

## 四、事务的隔离级别

- 并发控制机制可以解决并发问题。这使所有事务得以在彼此完全隔离的环境中运行
- 然而许多事务并不总是要求完全的隔离。如果允许降低隔离级别，则可以提高并发性

## 四、事务的隔离级别

### ■ SQL92标准定义了四种事务隔离级别

- **Note 1:** 隔离级别是针对连接（会话）而设置的，不是针对一个事务
- **Note 2:** 不同隔离级别影响读操作。**X**锁必须保持到事务结束

Oracle, MS  
SQL Server  
默认

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

MySQL默认

**Oracle只支持提交读和可串行读，MySQL和MS SQL Server都支持四种隔离级别**

## 四、事务的隔离级别

### ■ 未提交读（脏读） Read Uncommitted

- 允许读取当前数据页上的任何数据，不管数据是否已提交
- 事务不必等待任何锁，也不对读取的数据加锁
- 会出现丢失更新问题

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

## 四、事务的隔离级别

### ■ 提交读 Read Committed

- 保证事务不会读取到其他未提交事务所修改的数据（**可防止脏读**）
- 事务必须在所访问数据上加**S**锁，数据一旦读出，就马上释放持有的**S**锁
- 会出现丢失更新问题

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

## 四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level <b>READ COMMITTED</b>
2		Begin tran
3		Select SNAME from S Where SNAME='王红'
4	Begin tran	
5	Update s set SNAME='王红'	
6		Select SNAME from S Where SNAME='王红'
7	Commit tran	
8		Commit tran

等待

## 四、事务的隔离级别

### ■ 可重复读 Repeatable Read

- 保证事务在事务内部如果重复访问同一数据（记录集），数据不会发生改变。即，事务在访问数据时，其他事务不能修改正在访问的那部分数据
- 可重复读可以防止脏读和不可重复读取，但不能防止幻像
- 事务必须在所访问数据上加S锁，防止其他事务修改数据，而且S锁必须保持到事务结束
- 不会出现丢失更新

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否



## 四、事务的隔离级别

时间	连接1	连接2
1	<div>此两步执行</div>	Set transaction isolation level <b>REPEATABLE READ</b>
2		Begin tran
3	Begin tran	Select SNAME from S Where SNAME='王红'
4	Insert into s values(s08,'王红' ,23)	
5	Update s set age=22 where sname='张三'	
6	Update s set age=22 where sname='王红'	<div>出现幻象</div>
7	<div>此步须等待</div>	Select SNAME from S Where SNAME='王红'
8		Commit tran

## 四、事务的隔离级别

### ■ 可串行读 **Serializable**

- 保证事务调度是可串化的
- 事务在访问数据时，其他事务不能修改数据，也不能插入新元组
- 事务必须在所访问数据上加**S**锁，防止其他事务修改数据，而且**S**锁必须保持到事务结束
- 事务还必须锁住访问的整个表
- 不会出现丢失更新

隔离级别	脏读	不可重复读取	幻像
未提交读	是	是	是
提交读	否	是	是
可重复读	否	否	是
可串行读	否	否	否

## 四、事务的隔离级别

时间	连接1	连接2
1		Set transaction isolation level <b>SERIALIZABLE</b>
2		Begin tran
3	Begin tran	Select SNAME from S Where SNAME='王红'
4	Insert into s values(s08,'王红' ,23)	
5	此步须等待	Select SNAME from S Where SNAME='王红'
6		Commit tran

## 四、事务的隔离级别

- 不同隔离级别下**DBMS**加锁的动作有很大的差别

## 五、死锁(deadlock)

- **Two transactions each acquire a lock on a DB element the other needs**
- **Two transactions try to upgrade locks on elements the other is reading**

# 1、锁导致死锁

t	T1	T2
1	sL1(A)	
2		sL2(B)
3	Read(A)	Read(B)
4	xL1(B)	xL2(A)
5	Wait	Wait
6	Wait	Wait
7	wait	Wait
8	Wait	Wait
9	Wait	Wait
10	.....	.....

# 1、锁导致死锁

t	T1	T2
1	sL1(A)	
2		sL2(A)
3	Read(A)	Read(A)
4	A=A+B	
5	Upgrade(A)	A=A+100
6	Wait	Upgrade(A)
7	Wait	Wait
8	Wait	Wait
9	Wait	Wait
10	.....	.....

使用Update Lock

## 2、死锁的两种处理策略

- 死锁检测 **Deadlock Detecting**
  - 检测到死锁，再解锁
- 死锁预防 **Deadlock Prevention**
  - 提前采取措施防止出现死锁



# 3、Deadlock Detecting

## ■ Timeout 超时

- **Simple idea:** If a transaction hasn't completed in  $x$  minutes, abort it

## ■ Waiting graph 等待图

# 3、Deadlock Detecting

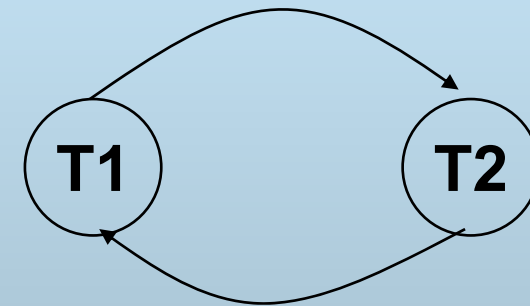
## ■ Waiting graph

- **Node: Transactions**
- **Arcs:  $T_i \rightarrow T_j$ ,  $T_i$ 必须等待 $T_j$ 释放所持有的某个锁才能继续执行**

如果等待图中存在环路,  
说明产生了死锁

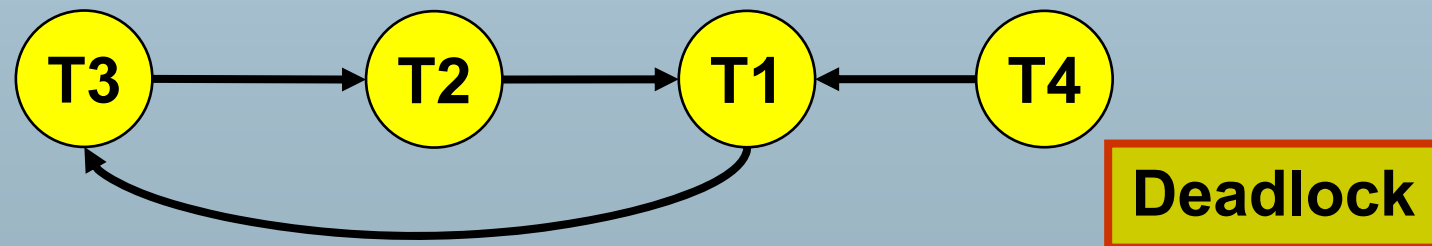
### 3、Deadlock Detecting

t	T1	T2
1	sL1(A)	
2		sL2(A)
3	Read(A)	Read(A)
4	sL1(B)	A=A+100
5	Read(B)	Upgrade(A)
6	A=A+B	Wait
7	Upgrade(A)	Wait
8	Wait	Wait
9	Wait	Wait
10	.....	.....



### 3、Deadlock Detecting

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(C);r2(C)		
3			L3(B);r3(B)	
4				L4(D);r4(D)
5		L2(A); wait		
6			L3(C);wait	
7				L4(A);wait
8	L1(B);wait			



# 4、Deadlock Prevention

- **方法1: Priority Order**
  - (按封锁对象的某种优先顺序加锁)
- **方法2: Timestamp**
  - (使用时间戳)

# 4、Deadlock Prevention

## ■ 方法1: Priority Order

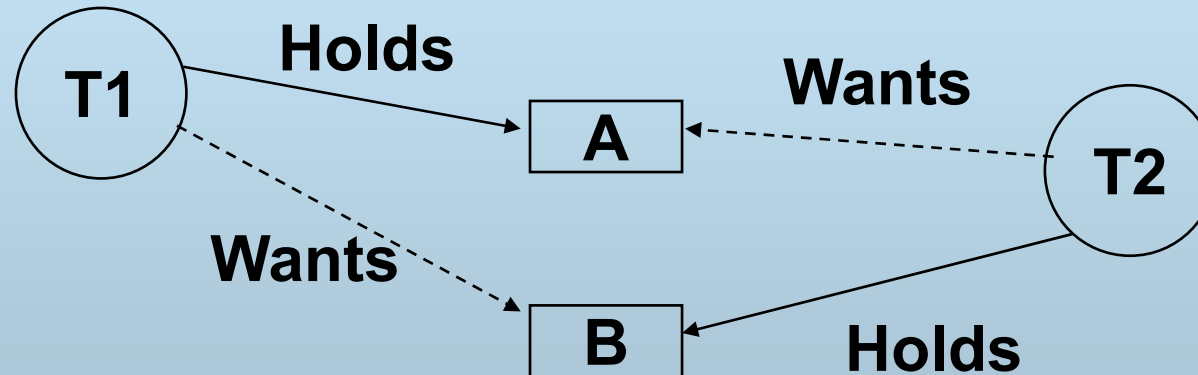
- 把要加锁的数据库元素按某种顺序排序
- 事务只能按照元素顺序申请锁

# 4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A);wait		
3			L3(B);r3(B)	
4				L4(A);wait
5			L3(C);w3(C)	
6			U3(B);U3(C)	
7	L1(B);w1(B)			
8	U1(A);U1(B)			
9		L2(A);L2(C)		
10	<div> <b>T1: A,B</b>  <b>T2: A,C</b>  <b>T3: B,C</b>  <b>T4: A,D</b> </div>	r2(C);w2(A)		
11		u2(A);u2(C)		
12				L4(A);L4(D)
13				r4(D);w4(A)
14				U4(A);U4(D)

## 4、Deadlock Prevention

### ■ 按序加锁可以预防死锁



**Impossible!**

**T2获得B上的锁之前，必须先要获得A上的锁**



# 4、Deadlock Prevention

## ■ 方法2: Timestamp

- 每个事务开始时赋予一个时间戳
- 如果事务T被Rollback然后再Restart, T的时间戳不变
- $T_i$ 请求被 $T_j$ 持有的锁, 根据 $T_i$ 和 $T_j$ 的timestamp决定锁的授予

## 4、Deadlock Prevention

### ■ Wait-Die Scheme 等待—死亡

#### ● T请求一个被U持有的锁

- ◆ If T is earlier than U then T **WAITS** for the lock
- ◆ If T is later than U then T **DIES** 【 *rollback* 】
  - We later restart T with its original timestamp

**Assumption:**

**$\text{timestamp}(T) < \text{timestamp}(U)$  means T is earlier than U**

# 4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A); <b>DIE</b>		
3			L3(B);r3(B)	
4				L4(A); <b>DIE</b>
5			L3(C);w3(C)	
6			U3(B);U3(C)	
7	L1(B);w1(B)			
8	U1(A);U1(B)			
9				L4(A);L4(D)
10		L2(A); <b>WAIT</b>		
11				r4(D);w4(A)
12				U4(A);U4(D)
13		L2(A);L2(C)		
14		r2(C);w2(A)		
15		u2(A);u2(C)		

# 4、Deadlock Prevention

## ■ Wound-Wait Scheme 伤害—等待

### ● T请求一个被U持有的锁

◆ If T is earlier than U then T **WOUNDS** U

● U must release its locks then rollback and restart and the lock is given to T

◆ If T is later than U then T **WAITS** for the lock

# 4、Deadlock Prevention

t	T1	T2	T3	T4
1	L1(A); r1(A)			
2		L2(A); WAIT		
3			L3(B); r3(B)	
4				L4(A); WAIT
5	L1(B); w1(B)		WOUNDED	
6	U1(A); U1(B)			
7		L2(A); L2(C)		
8		r2(C); w2(A)		
9		u2(A); u2(C)		
10				L4(A); L4(D)
11				r4(D); w4(A)
12				U4(A); U4(D)
13			L3(B); r3(B)	
14			L3(C); w3(C)	
15			U3(B); U3(C)	

# 4、Deadlock Prevention

## ■ Comparison

### ● Wait-Die:

- ◆ **Rollback**总是发生在请求锁阶段，因此要**Rollback**的事务操作比较少，但**Rollback**的事务数会比较多

### ● Wound-Wait:

- ◆ 发生**Rollback**时，要**Rollback**的事务已经获得了锁，有可能已经执行了较长时间，因此**Rollback**的事务操作会较多，但**Rollback**的事务数预期较少，因为可以假设事务开始时总是先请求锁
- ◆ 请求锁时**WAIT**要比**WOUND**要更普遍，因为一般情况下一个新事务要请求的锁总是被一个较早的事务所持有

## 4、Deadlock Prevention

### ■ Why wait-die and wound-wait work?

- 假设  $T1 \rightarrow T2 \rightarrow \dots \rightarrow Tk \rightarrow T1$  【*deadlock*】
- 在wait-die scheme中，只有当  $T_i < T_j$  时才会有  $T_i \rightarrow T_j$ ，因此有
  - ◆  $T1 < T2 < \dots < Tk < T1$  -- Impossible!
- 在wound-wait scheme中，只有当  $T_i > T_j$  时才会有  $T_i \rightarrow T_j$ ，因此有
  - ◆  $T1 > T2 > \dots > Tk > T1$  -- Still impossible!

# 再论“锁机制”

## ■ 两种并发控制思路

### ● 悲观并发控制 --- “悲观锁”

- ◆ 立足于事先预防事务冲突
- ◆ 采用锁机制实现，事务访问数据前都要申请锁
- ◆ 锁机制影响性能，容易带来死锁、活锁等副作用

### ● 乐观并发控制

- ◆ 乐观并发控制假定不太可能（但不是不可能）在多个用户间发生资源冲突，允许不锁定任何资源而执行事务。只有试图更改数据时才检查资源以确定是否发生冲突。如果发生冲突，应用程序必须读取数据并再次尝试进行更改。



# 乐观并发控制

## ■ 动机

- 如果大部分事务都是只读事务，则并发冲突的概率比较低；即使不加锁，也不会破坏数据库的一致性；加锁反而会带来事务延迟

# 乐观并发控制

## ■ 在ADO程序中

### ● Recordset打开时指定LockType

- ◆ 0(adLockReadOnly): recordset的记录为只读
- ◆ 1(adLockPessimistic): 悲观并发控制, 只要保持Recordset为打开, 别人就无法编辑该记录集中的记录.
- ◆ 2(adLockOptimistic): “乐观” 并发控制, 当update recordset中的记录时, 将记录加锁
- ◆ 3(adLockBatchOptimistic): 以批模式时更新记录时加锁

# 乐观并发控制

- 基于事后协调冲突的思想，用户访问数据时不加锁；如果发生冲突，则通过回滚某个冲突事务加以解决
- 由于不需要加锁，因此开销较小，并发度高
- 但需要确定哪些事务发生了冲突
  - 使用“有效性确认(Validation)”

# 乐观并发控制

## ■ 有效性确认协议

### ● 每个更新事务Ti在其生命周期中按以下三个阶段顺序执行

- ◆ 读阶段：数据被读入到事务Ti的局部变量中。此时所有 **write** 操作都针对局部变量，并不对数据库更新
- ◆ 有效性确认阶段：Ti进行有效性检查，判定是否可以将 **write** 操作所更新的局部变量值写回数据库而不违反可串行性
- ◆ 写阶段：若Ti通过有效性检查，则进行实际的写数据库操作，否则回滚Ti

# 乐观并发控制

- 有效性检查方法（第二阶段）
  - 基于时间戳（行版本）的方式
    - ◆ Version --- MySQL
    - ◆ Timestamp --- MS SQL Server, Oracle

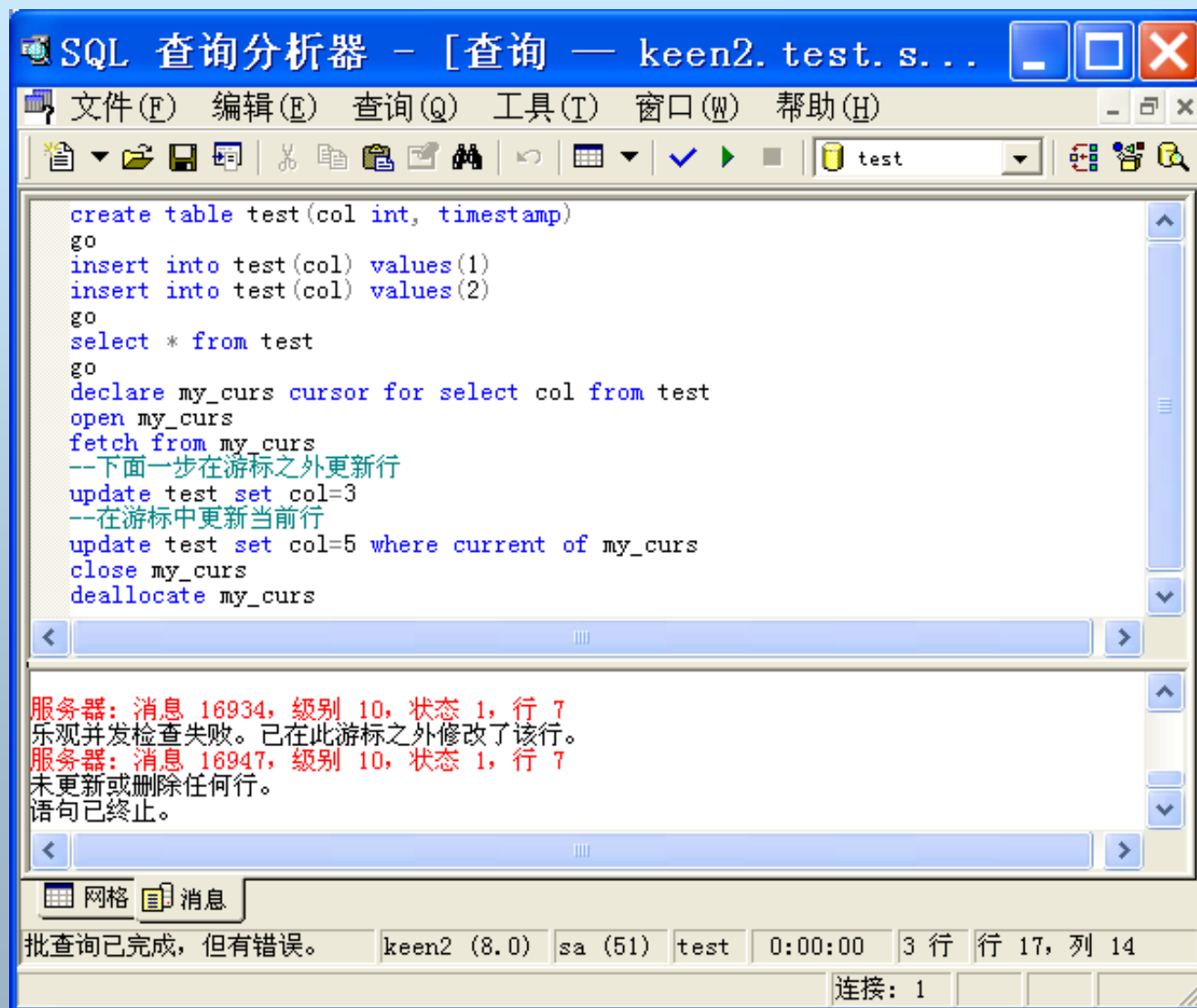
# 乐观并发控制

- 基于行版本的乐观并发控制
  - **MS SQL Server**使用特殊数据类型**timestamp**（数据库范围内唯一的8字节二进制数）
  - 全局变量**@@DBTS**返回当前数据库最后所使用的时间戳值
  - 如果一个表包含 **timestamp** 列，则每次由 **INSERT**、**UPDATE** 或 **DELETE** 语句修改一行时，此行的 **timestamp** 值就被置为当前的 **@@DBTS** 值，然后 **@@DBTS** 加1
  - 服务器可以比较某行的当前**timestamp**和游标提取时的**timestamp**值，确定是否更新

# 乐观并发控制

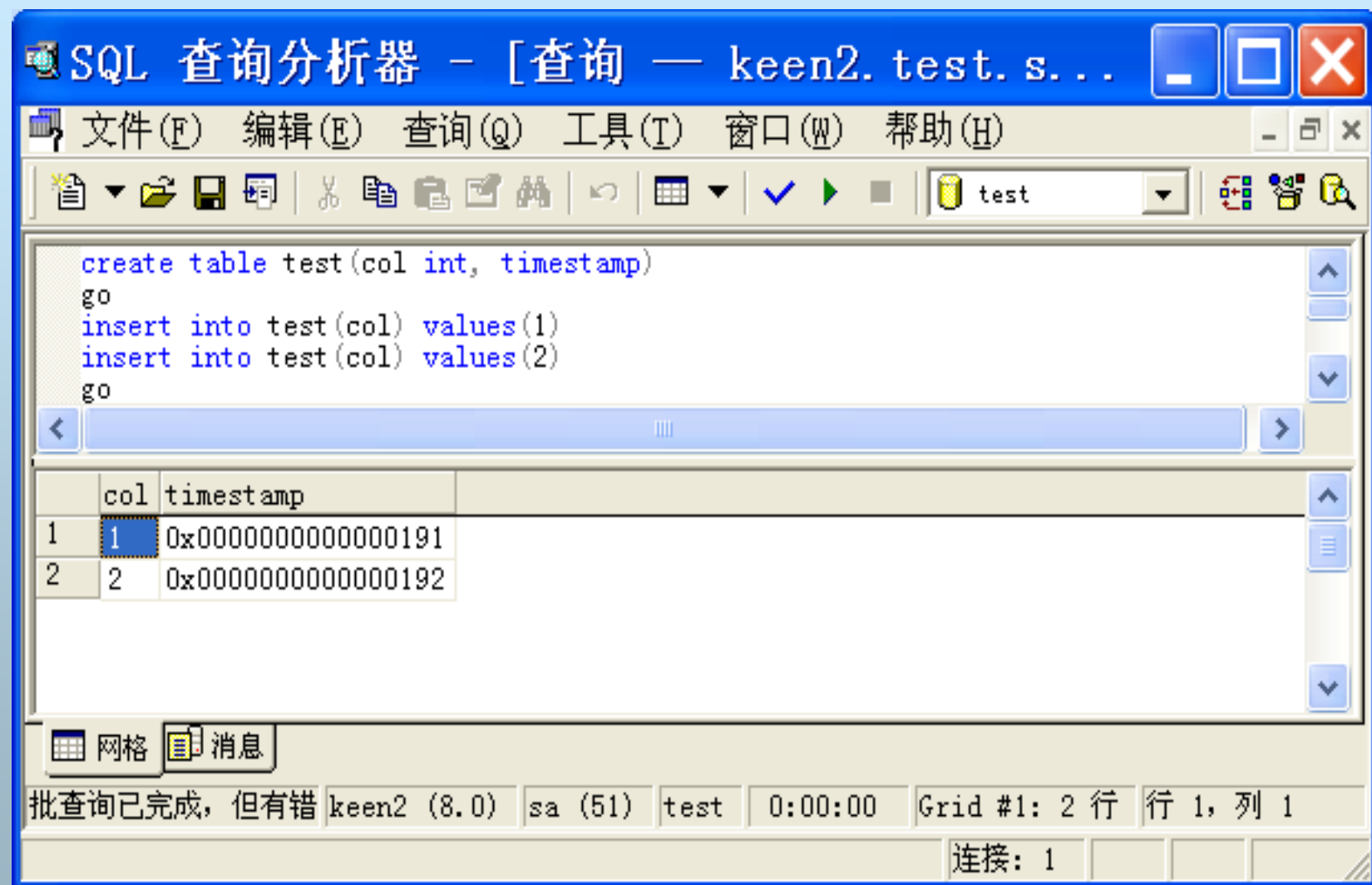
- 基于行版本的乐观并发控制
  - 当用户打开游标时，**SQL Server**保存行的当前**timestamp**；当在游标中想更新一行时，**SQL Server**为更新数据自动添加一条**Where**子句
    - ◆ **WHERE timestamp列 <= <old timestamp>**
  - 如果不相等，则报错并回滚事务

# 乐观并发控制





# 乐观并发控制



# 本章小结

- 并发操作问题
- 调度与可串行性
  - 可串行化调度
  - 冲突可串行性及判断
- 锁与可串行性实现
  - 2PL
  - 多种锁模式：X、S、U
  - 多粒度锁与意向锁
- 事务的隔离级别
- 死锁