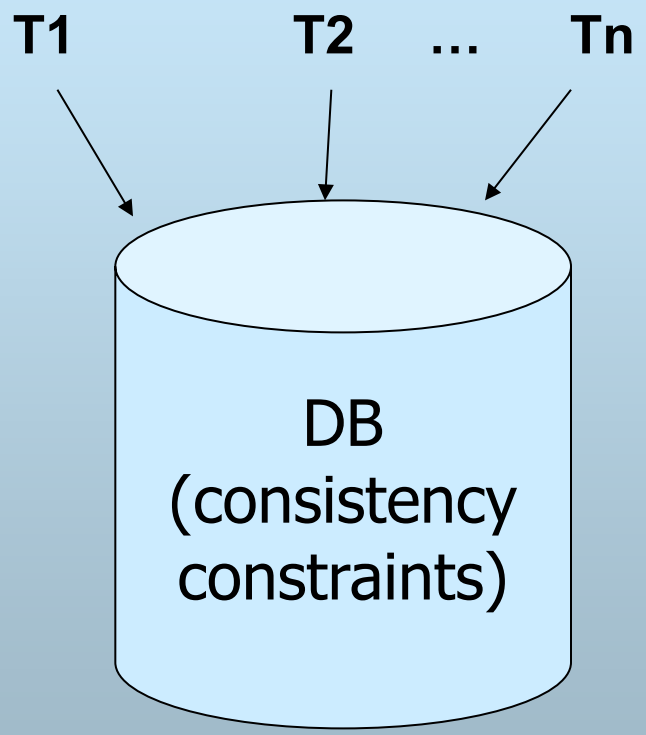
Transaction Processing (II)
Concurrency Control

金培权（jpq@ustc.edu.cn）

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
  - 数据库的恢复技术 [Chp. 17]
    - Deal with failure
  - 并发控制技术 [Chp. 18 & 19]
    - Deal with data sharing
  - 完整性控制技术
    - Enable constraints
  - 安全性控制技术
    - Authorization and authentication

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?

- 丢失更新 Lost update
- 脏读 Dirty read
- 不一致分析 Inconsistent analysis
  - 不可重复读 Nonrepeatable read
  - 幻像读 Phantom read

主要内容

- 并发事务调度与可串性 (Scheduling and Serializability)
- 锁与可串性实现 (Locks)
- 乐观并发控制 (Optimistic CC)

一、并发调度

- 并发调度（简称“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)
Schedule = r1(A) w1(A) r2(A) w2(A) r1(B) w1(B) r2(B) w2(B)

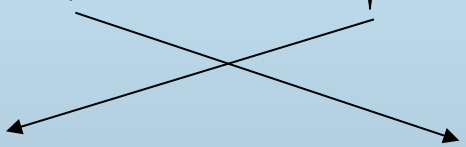
- Note
  - 调度中每个事务的读写操作保持原来顺序
  - 事务调度时不考虑
    - 数据库的初始状态 (Initial state)
    - 事务的语义 (Transaction semantics)

一、并发调度

- 多个事务的并发执行存在多种调度方式

Example:

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



Sb = 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?

一、并发调度

- 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 ... Tn
- Serializable Schedule（可串化调度）
  - 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

一、并发调度

- 可串化调度的正确性
  - Correctness of DB: 单个事务的执行保证DB从一个一致性状态变化到另一个一致性状态
  - N个事务串行调度执行仍保证 Correctness of DB

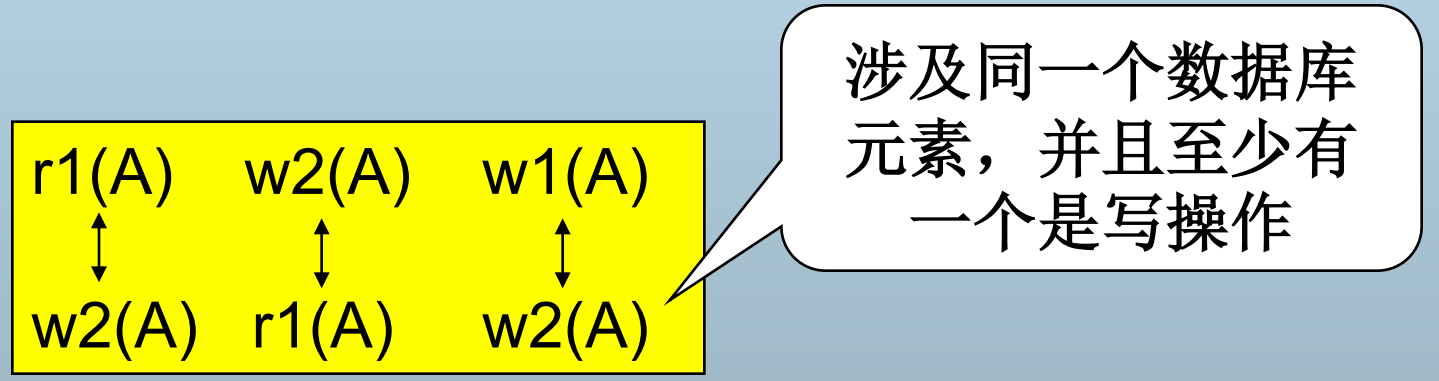
一、并发调度

- **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?** ⇓ **冲突可串性**

二、冲突可串性

- **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**



1、冲突操作

- **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

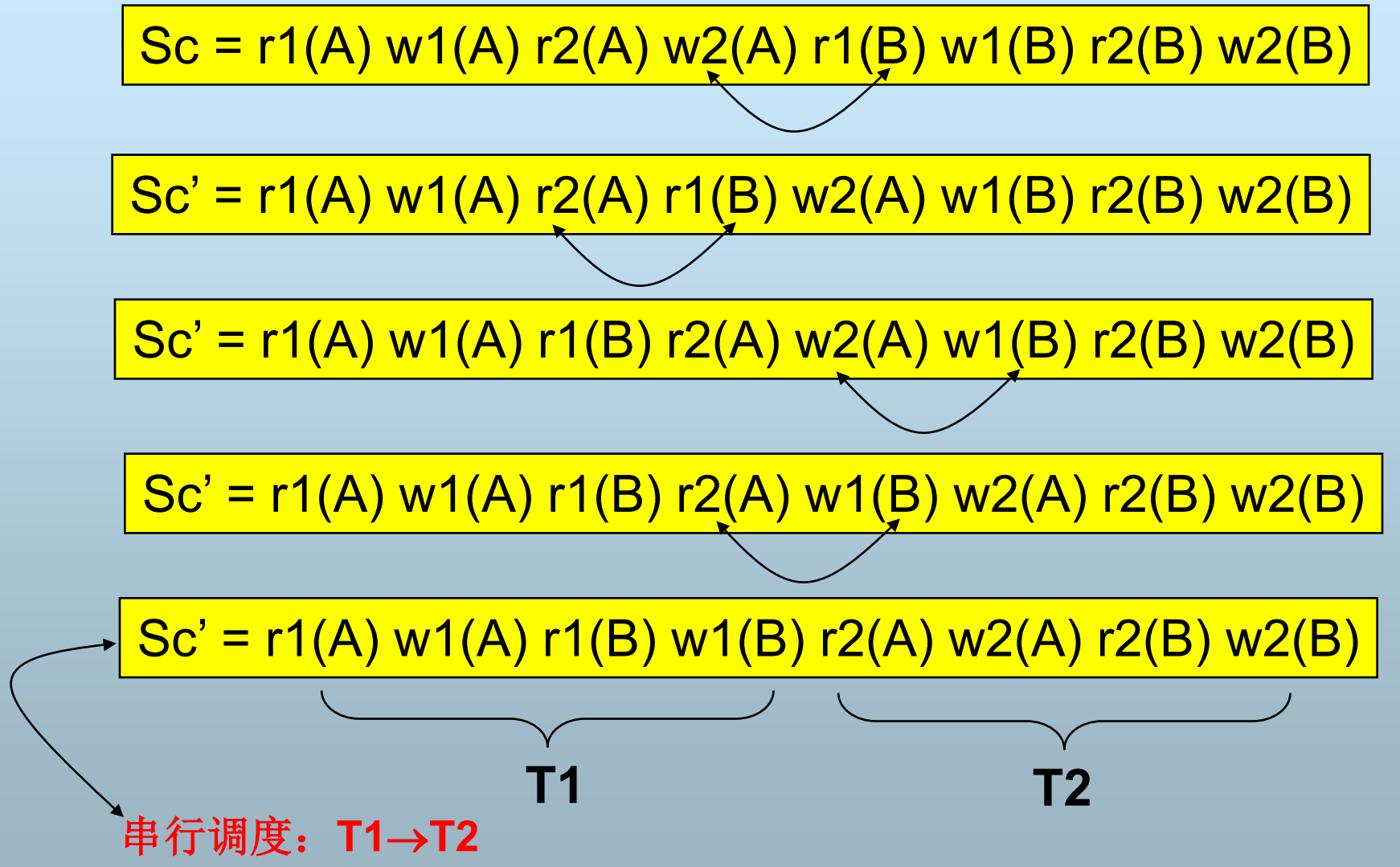
1、冲突操作

Schedule C

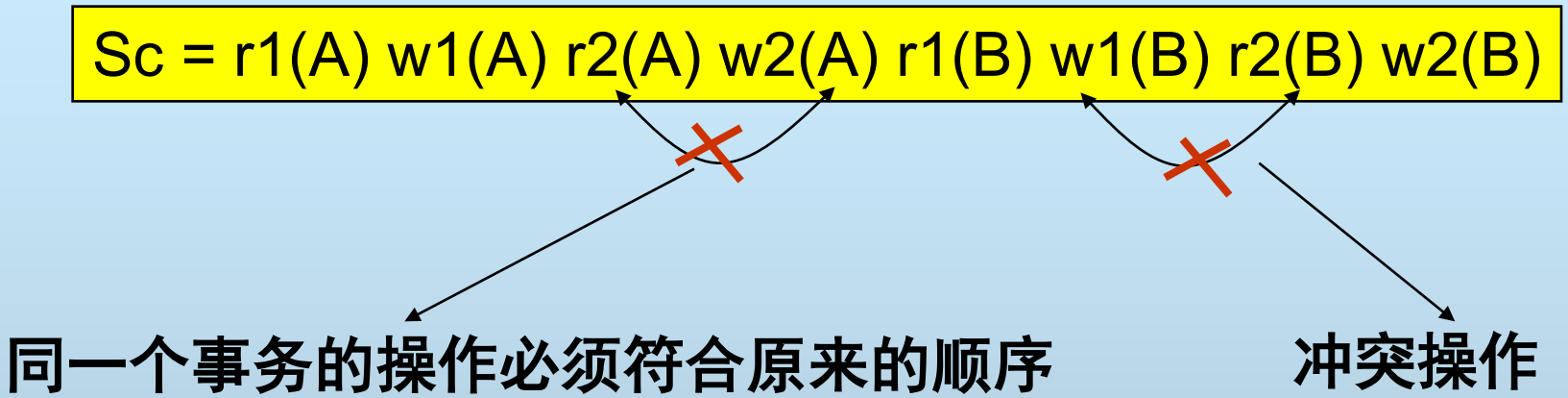
T1	T2	A	B
Read(A, t); t ← t+100		25	25
Write(A, t);		125	25
	Read(A, s); s ← s×2;		
	Write(A, s);	250	25
Read(B, t);			
t ← t+100;			
Write(B, t);		250	125
	Read(B, s); s ← s×2;		
	Write(B, s);	250	250

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

1、冲突操作


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
串行调度：T1→T2

1、冲突操作



2、冲突可串

- 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.

2、冲突可串

- 定理
  - 如果一个调度满足冲突可串性，则该调度是可串化调度

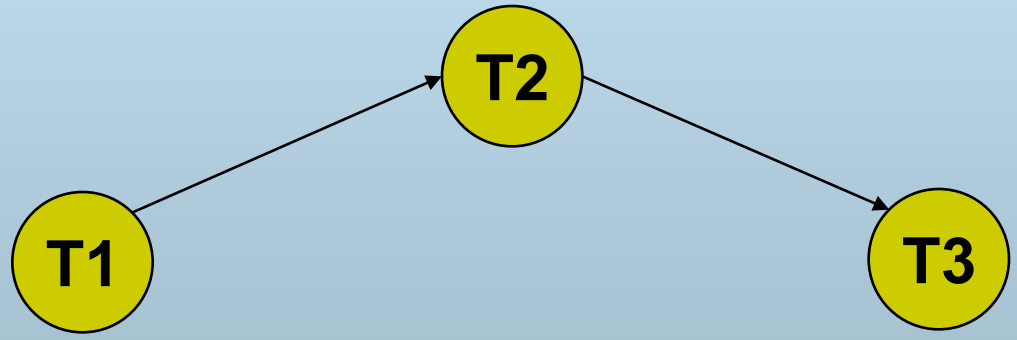
3、优先图 (Precedence Graph)

- 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): Ti → Tj ，满足 Ti $<_s$ Tj
    - 存在Ti中的操作A1和Tj中的操作A2，满足
      - A1在A2前，并且
      - A1和A2是冲突操作

3、优先图

Example

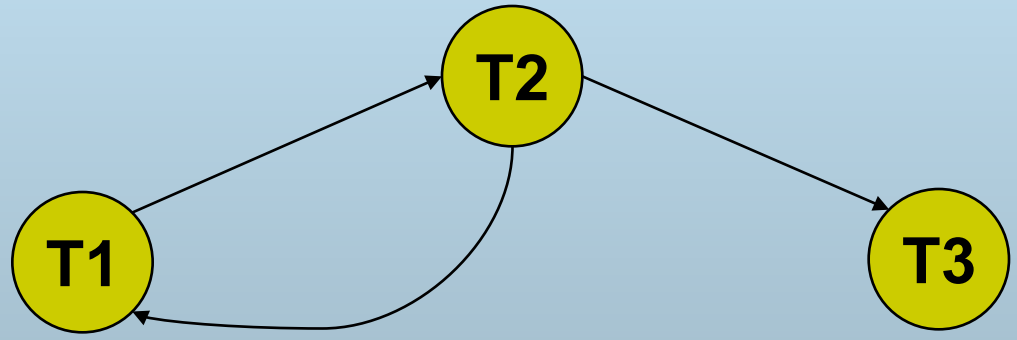
S = r2(A) r1(B) w2(A) r3(A) w1(B) w3(A) r2(B) w2(B)



3、优先图

Example

S = r2(A) r1(B) w2(A) r2(B) r3(A) w1(B) w3(A) w2(B)



3、优先图

- 优先图与冲突可串性
  - 给定一个调度S，构造S的优先图P(S)，若P(S)中无环，则S满足冲突可串性
  - 证明：归纳法, see chp.18

三、视图可串性

- 弱于冲突可串性，但仍可保证调度的可串性

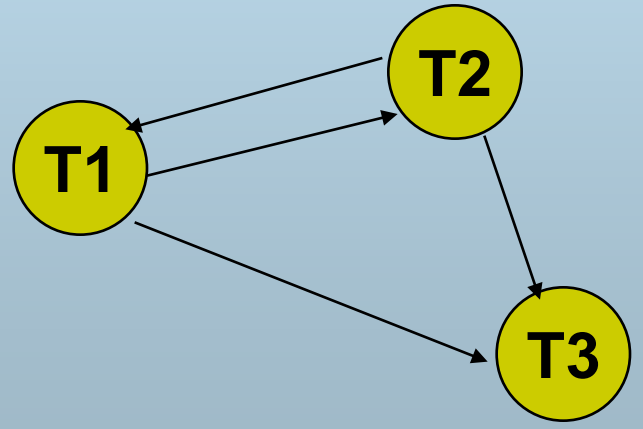
Conflict equivalent	View equivalent
⇕	⇕
Conflict serializable	View serializable

1、View Equivalent

T1:			r1(A)		w1(B)		
T2:	r2(B)	w2(A)				w2(B)	
T3:				r3(A)			w3(B)

S1＝r2(B) w2(A) r1(A) r3(A) w1(B) w2(B) w3(B)

Precedence graph



Not conflict serializable

1、View Equivalent

S1

```
T1:                  r1(A)        w1(B)
T2:  r2(B)  w2(A)                          w2(B)
T3:                        r3(A)                    w3(B)
```

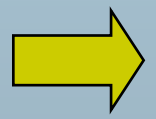
S2 (serial)

```
T1:                                    r1(A)  w1(B)
T2:  r2(B)  w2(A) w2(B)
T3:                                                  r3(A)     w3(B)
```

In S1,S2:

Both T1 read same A,B
Both T2 read same A,B
Both T3 read same A,B

Final state:
A: T2 writes
B: T3 writes



S1可串！

1、View Equivalent

- 如果按照conflict serializability，S1不可串。实际上S1可串
  - Conflict serializability 过于严格
  - View serializability 解决这一问题

1、View Equivalent

Definition S1和S2中任何事务的读操作的源都相同

- 调度S1和S2视图等价，如果
  - (1) If in S1: wj(A) ⇒ ri(A)
    then in S2: wj(A) ⇒ ri(A)
  - (2) If in S1: ri(A) reads the initial DB value,
    then in S2: ri(A) also reads the initial DB value
  - (3) If in S1: Ti does the last write on A,
    then in S2: Ti also does the last write on A

wj(A) ⇒ ri(A): wj(A) ..... ri(A)
No w(A)

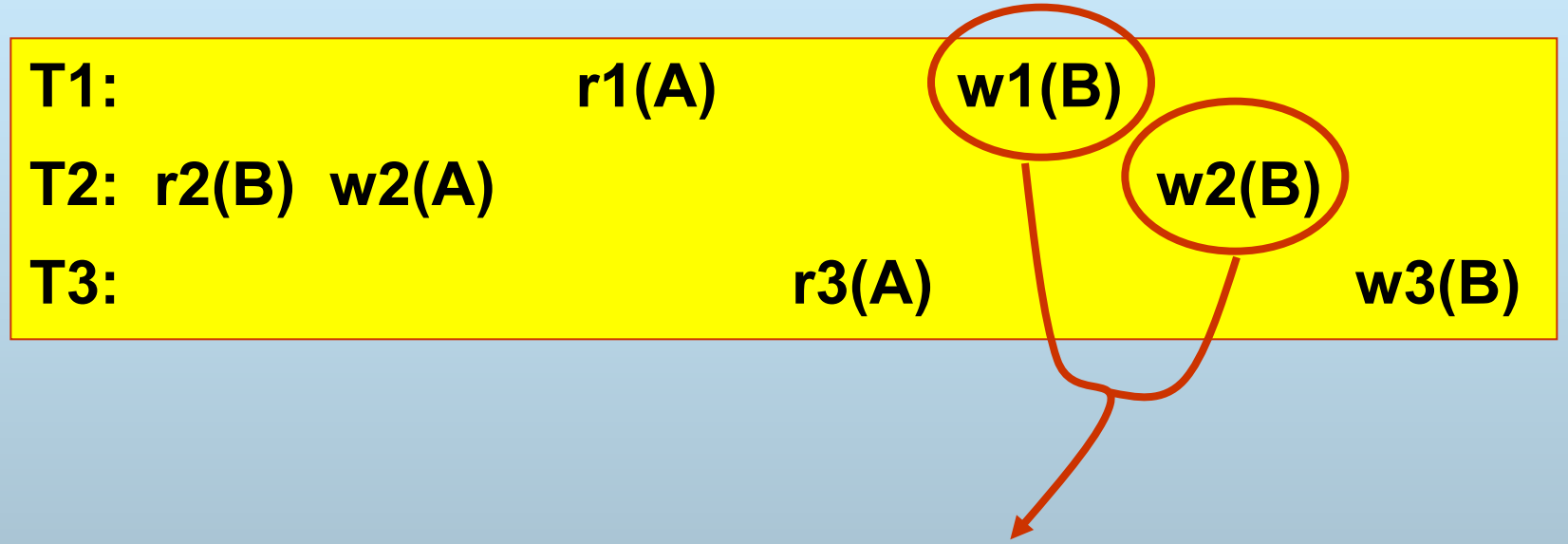
1、View Equivalent

Definition

- **Schedule S1 is View Serializable if it is view equivalent to some serial schedule**

2、视图可串 vs. 冲突可串

- **Difference**



Can not swap in conflict serializability!
Can swap in view serializability!

3、Polygraph

- 视图可串性判断: 多重图 (Polygraph)
  - 对于调度Z，其Polygraph为LP(Z)

3、Polygraph

- Node：事务＋假想的事务Tb和Tf
- Arcs
  - 事务（包括Tb和Tf）之间

(1) Tb对所有DB元素执行写操作，构成DB初始状态

(2) Tf读所有DB元素，得到DB终态

3、Polygraph

(3) 建立Arcs

(3a) If $w_i(A) \Rightarrow r_j(A)$ in S, add $T_i \xrightarrow{0} T_j$

3、Polygraph

(3b) For each $w_i(A) \Rightarrow r_j(A)$ do

consider each $w_k(A)$: $[T_k \neq T_b]$ $k \neq i, j$

- If $T_i \neq T_b \wedge T_j \neq T_f$ then insert

$$\begin{cases} T_k \xrightarrow{p} T_i & \text{some new p} \\ T_j \xrightarrow{p} T_k \end{cases}$$

- If $T_i = T_b \wedge T_j \neq T_f$ then insert

$$T_j \xrightarrow{0} T_k$$

- If $T_i \neq T_b \wedge T_j = T_f$ then insert

$$T_k \xrightarrow{0} T_i$$

3、Polygraph

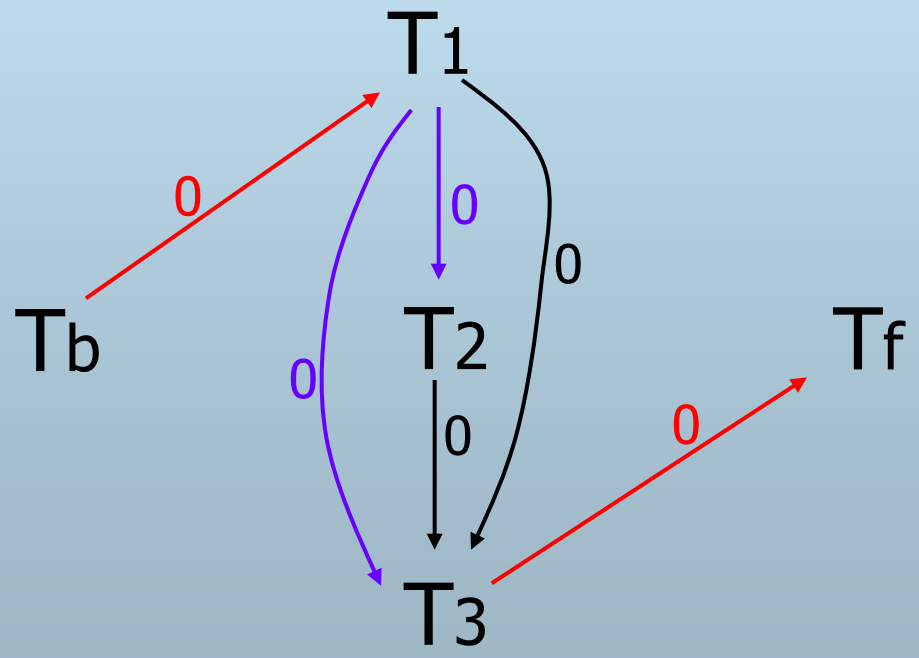
(3c) 对于每一对 Ti $\overset{p}{\rightarrow}$ Tj, 选择其中一个，将其在Polygraph中删除，如果能使Polygraph成为无环图，则调度S是视图可串化的 (V-S)

3、Polygraph

Example check if Q is V-S:

Q = r1(A) w2(A) w1(A) w3(A)

Q' = wb(A) ⇒ r1(A) w2(A) w1(A) w3(A) ⇒ rf(A)



rule 3(a)

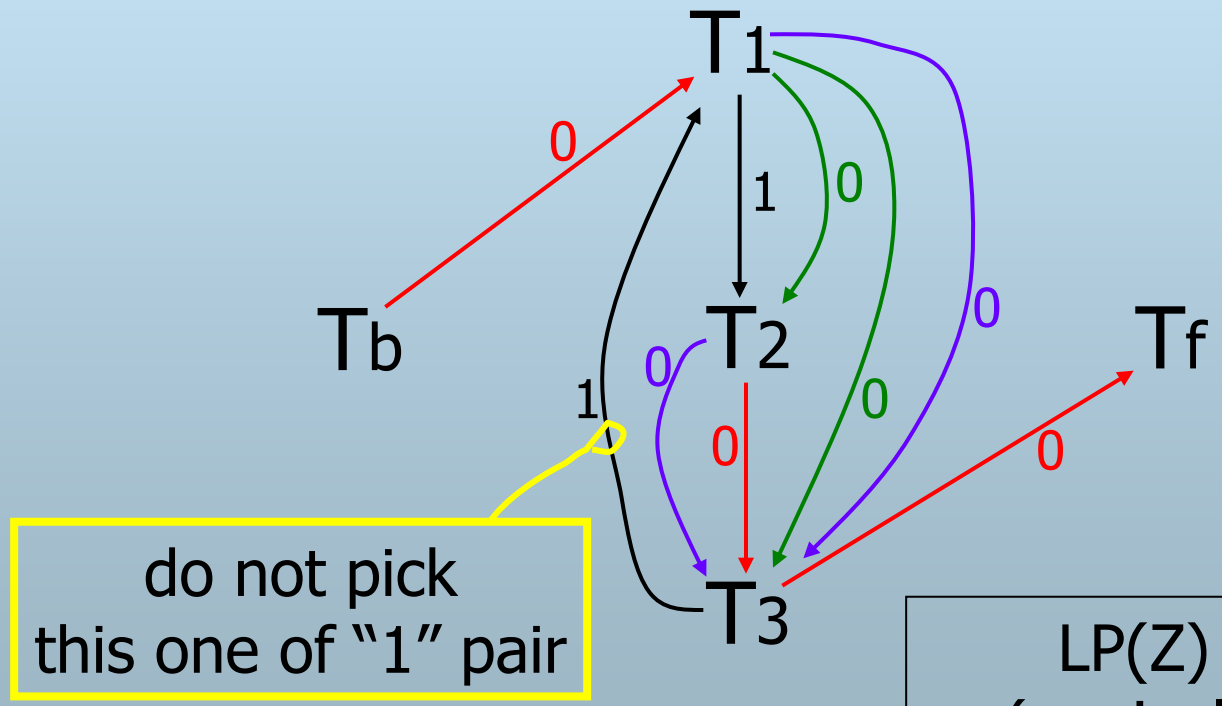
rule 3(b)

rule 3(b)

3、Polygraph

Example

Z=wb(A) ⇒ r1(A)w2(A) ⇒ r3(A) w1(A) w3(A) ⇒ rf(A)



rule 3(a)

rule 3(b)

rule 3(b)

rule 3(c)

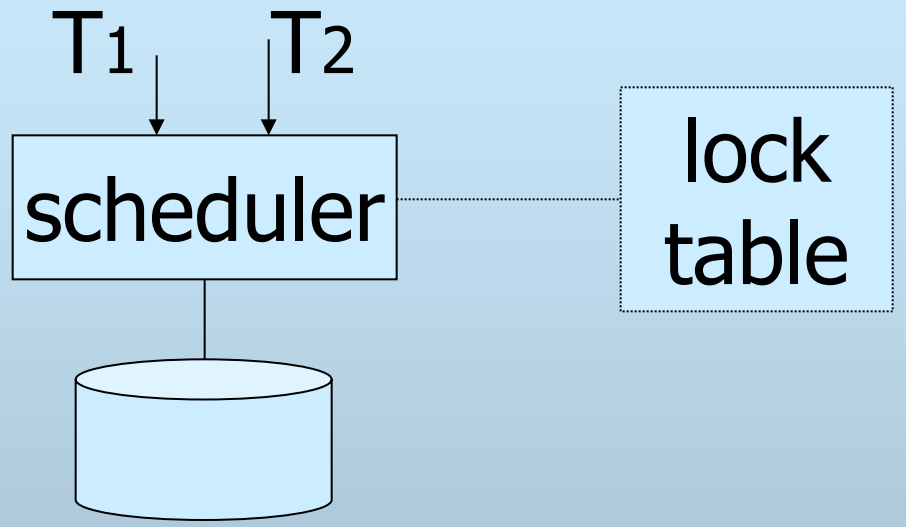
LP(Z) acyclic, so Z is V-S
(equivalent to Tb T1 T2 T3 Tf)

四、锁机制

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

给定n个并发事务，确定一个可串化调度

1、锁简介



Two new actions:

lock (exclusive): l_i (A)

unlock: u_i (A)

2、两阶段锁(2PL)

- **Two Phase Locking**

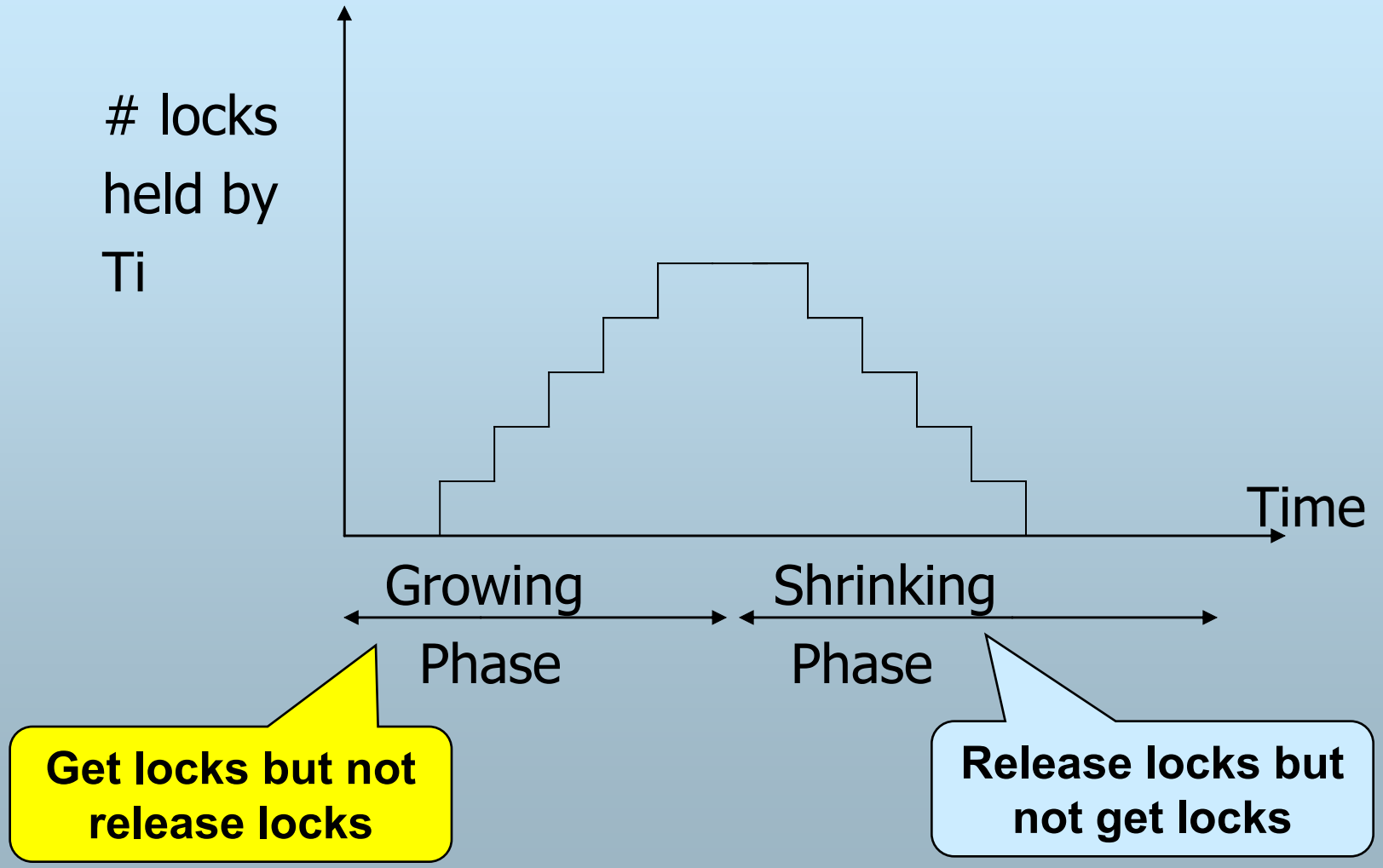
T_i = ... $l_i(A)$ ..... $u_i(A)$ .....

no unlocks　　no locks

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

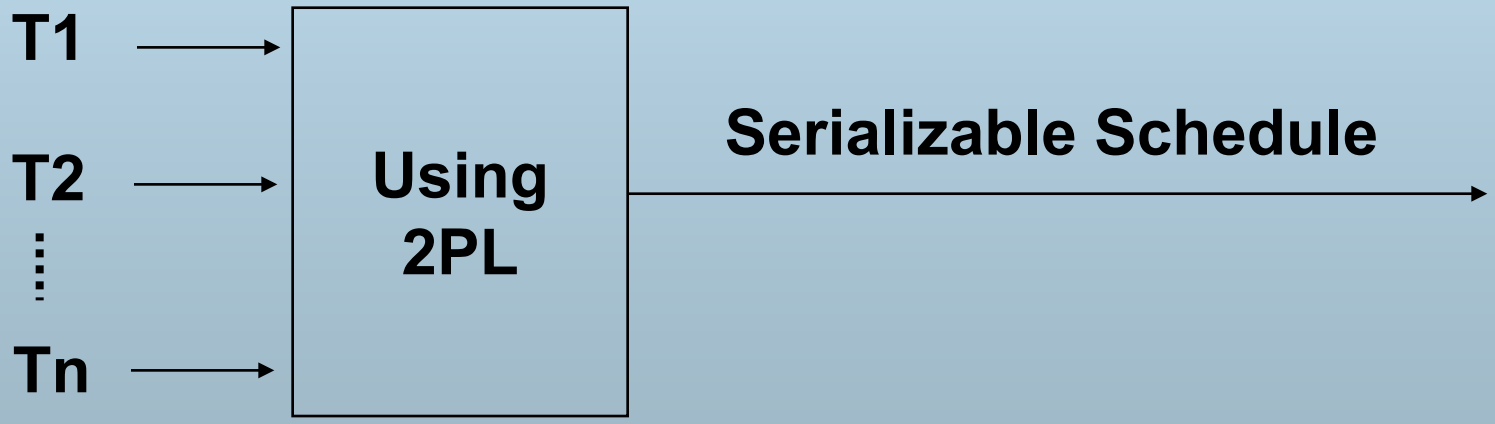
Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, Irving L. Traiger:
The Notions of Consistency and Predicate Locks in a Database System. Commun. ACM 19(11): 624-633 (1976)

2、两阶段锁(2PL)


locks
held by
Ti
Time
Growing
Phase
Shrinking
Phase
Get locks but not release locks
Release locks but not get locks

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，则该调度是冲突可串化调度



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
      - read之前必须先获得S锁
      - 多个事务可以同时持有一个数据上的S锁 (share lock)
    - X Lock
      - write之前必须先获得X锁，若已持有S锁，则upgrade
      - 一个数据上的X锁永远只能被一个事务持有 (exclusive)
    - Update Lock
    - ……

3、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	……	……

3、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

3、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?

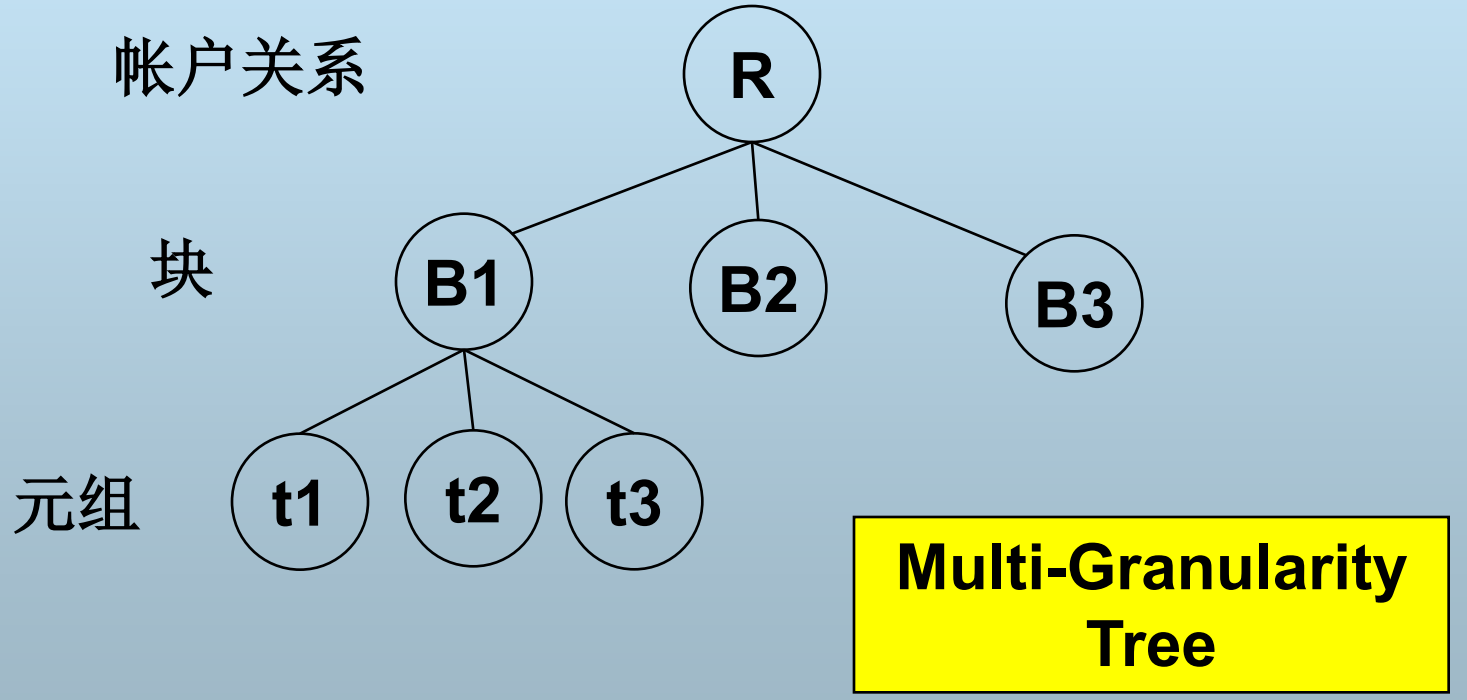
- 调度与可串性
- 锁与可串性实现
  - 2PL
    - S Lock
    - X Lock
    - U Lock
  - Multi-granularity Lock 多粒度锁
  - Intension Lock 意向锁

4、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**

4、Multi-Granularity Lock

- 多粒度锁：同时支持多种不同的锁粒度



4、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

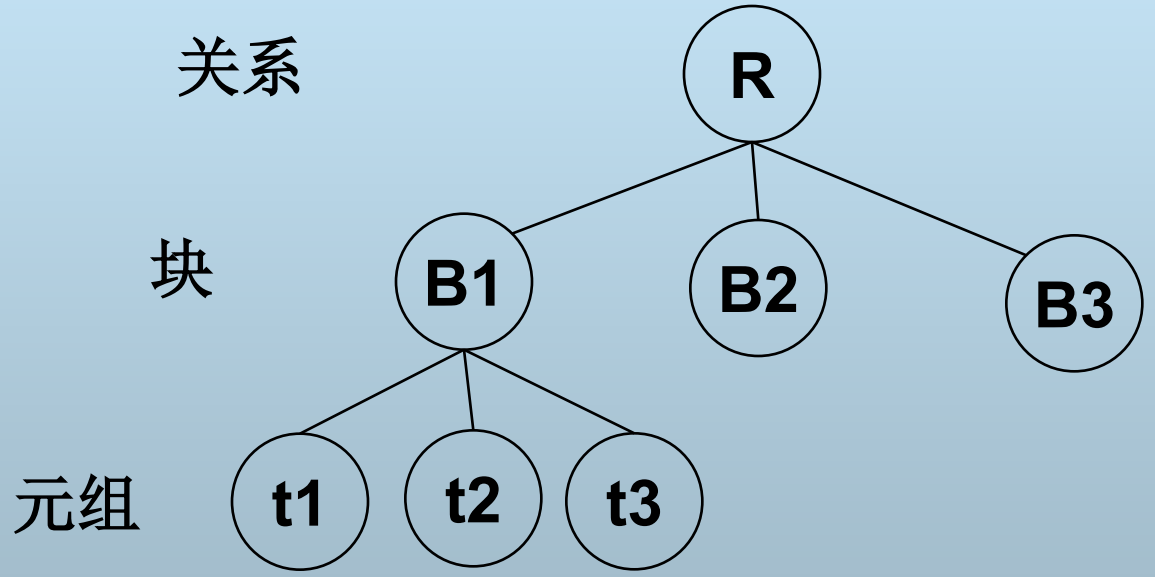
4、Multi-Granularity Lock

- 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

4、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) 解决此问题

5、Intension Lock



5、Intension Lock

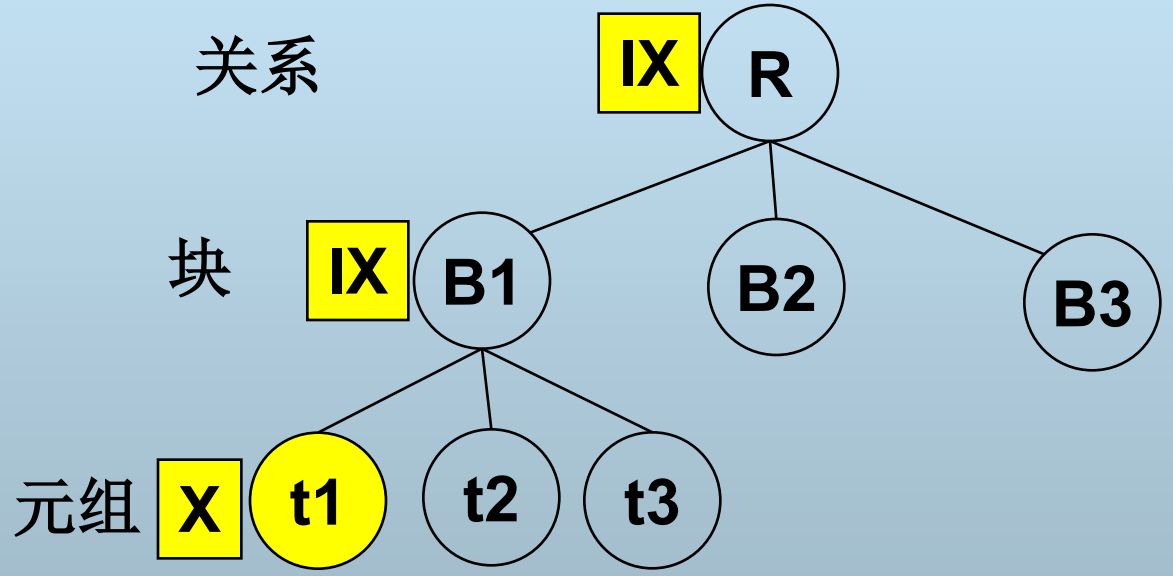
- IS锁（Intent Share Lock，意向共享锁，意向读锁）
- IX锁（Intent Exlusive Lock，意向排它锁，意向写锁）

5、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

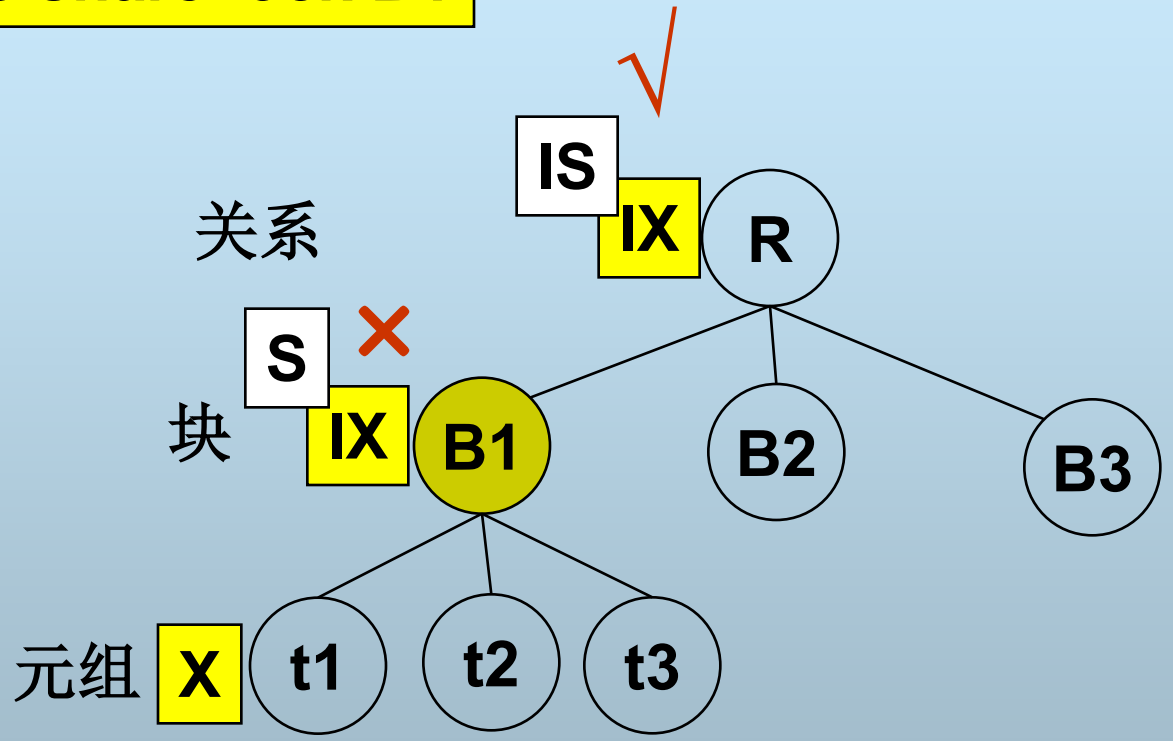
5、Intension Lock

Want to exclusively lock t1

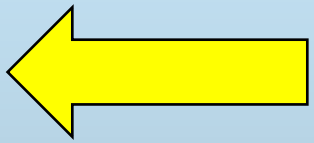


5、Intension Lock

Want to share lock B1



Where are we?

- 并发事务调度与可串性 (Scheduling and Serializability)
- 锁与可串性实现 (Locks)
- 乐观并发控制 (Optimistic CC) 

再论并发控制

- 哪些并发操作可能是冲突的？
  - 读-读 😀
  - 读-写 😢
  - 写-读 😢
  - 写-写 😢

  } 锁机制下都需要加锁影响并发性能

- 代价
  - 加锁代价
  - 锁表存储代价
  - 死锁、活锁

  } 能否不用锁？

六、乐观并发控制

- 两种并发控制思路
  - 乐观并发控制 --- “乐观锁”
    - 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。
    - 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
    - “读不加锁，写时协调”
  - 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

六、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
- 由于读不需要加锁，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)”

六、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

六、乐观并发控制

- 有效性检查方法（第二阶段）
  - 基于行版本的方式
    - Version --- MySQL
    - Timestamp --- MS SQL Server, Oracle
  - 基于值比较的方式
    - MS SQL Server

六、乐观并发控制

- 基于行版本的乐观并发控制 ——以MS SQL Server为例
  - MS SQL Server允许在游标中使用乐观并发控制

SQL Server支持的游标选项：

READ_ONLY
OPTIMISTIC WITH VALUES: 基于值比较的乐观并发控制
OPTIMISTIC WITH ROW VERSIONING: 基于时间戳的乐观并发控制
SCROLL_LOCKS: 悲观并发控制

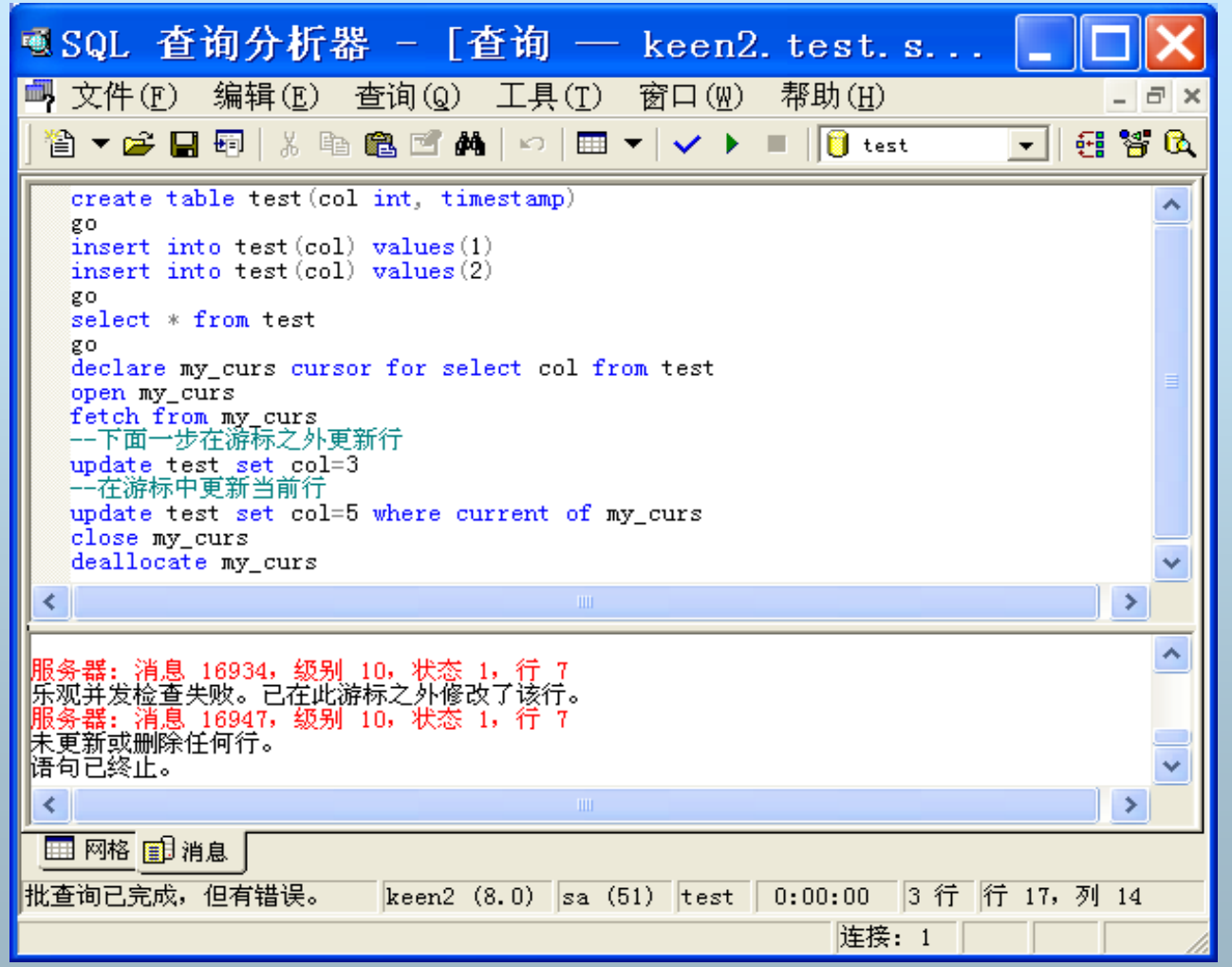
六、乐观并发控制

- 基于行版本的乐观并发控制 ——以MS SQL Server为例
  - 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

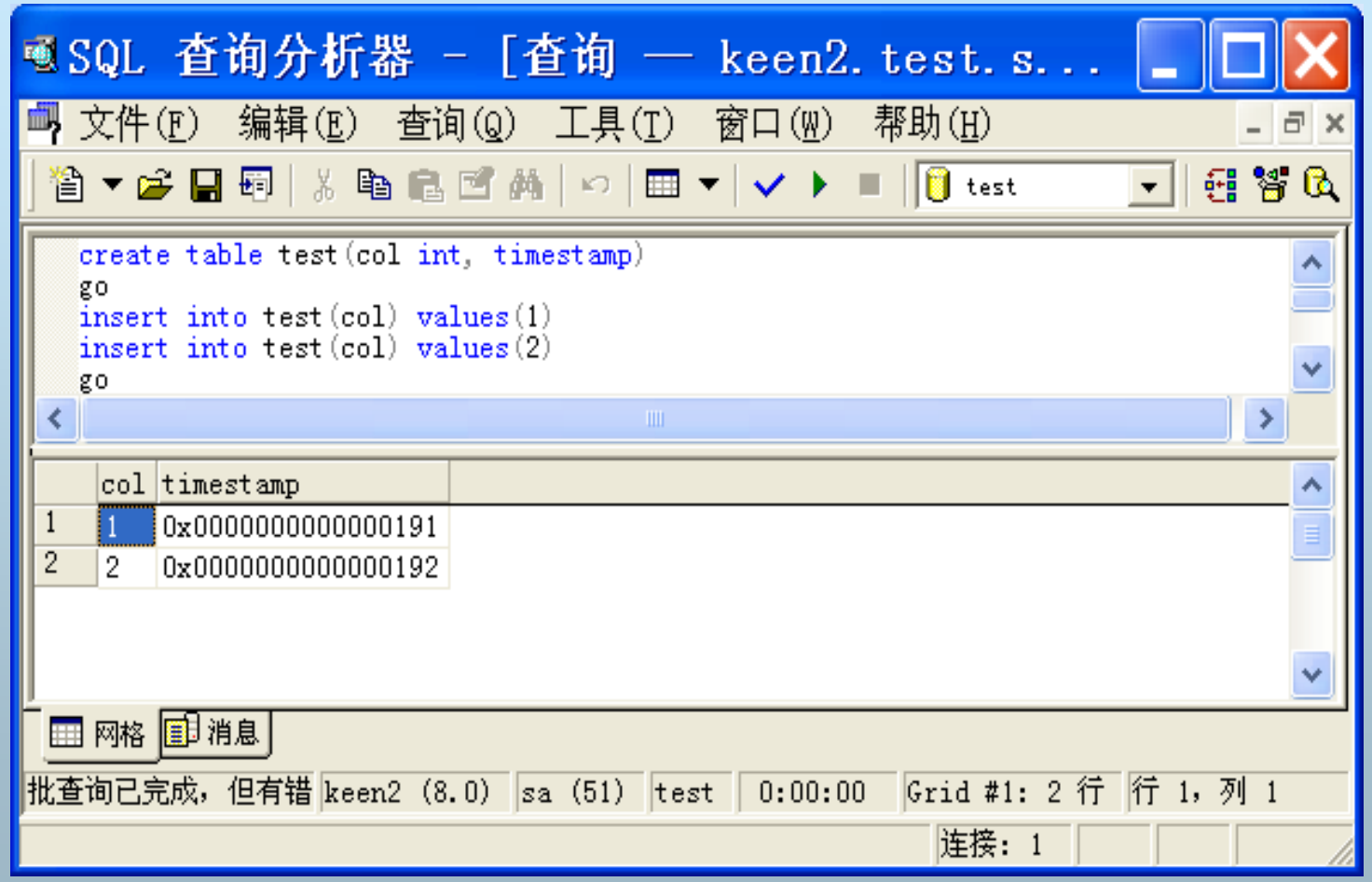
六、乐观并发控制

- 基于行版本的乐观并发控制 ——以MS SQL Server为例
  - 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

六、乐观并发控制


SQL 查询分析器 - [查询 — keen2.test.s...
文件(F) 编辑(E) 查询(Q) 工具(T) 窗口(W) 帮助(H)
test
create table test(col int, timestamp)
go
insert into test(col) values(1)
insert into test(col) values(2)
go
select * from test
go
declare my_curs cursor for select col from test
open my_curs
fetch from my_curs
--下面一步在游标之外更新行
update test set col=3
--在游标中更新当前行
update test set col=5 where current of my_curs
close my_curs
deallocate my_curs
服务器: 消息 16934, 级别 10, 状态 1, 行 7
乐观并发检查失败。已在此游标之外修改了该行。
服务器: 消息 16947, 级别 10, 状态 1, 行 7
未更新或删除任何行。
语句已终止。
网格
消息
批查询已完成，但有错误。
keen2 (8.0)
sa (51)
test
0:00:00
3 行
行 17, 列 14
连接: 1

六、乐观并发控制


SQL 查询分析器 - [查询 — keen2.test.s...
文件(F) 编辑(E) 查询(Q) 工具(T) 窗口(W) 帮助(H)
test
create table test(col int, timestamp)
go
insert into test(col) values(1)
insert into test(col) values(2)
go
col timestamp
1 1 0x0000000000000191
2 2 0x0000000000000192
网格 消息
批查询已完成，但有错 keen2 (8.0) sa (51) test 0:00:00 Grid #1: 2 行 行 1，列 1
连接: 1

六、乐观并发控制

- 基于值比较的乐观并发控制——MS SQL Server
  - 如果表中没有timestamp列，SQL Server在游标并发中将采用基于值比较的方式
  - 如果用户试图修改某一行，则此行的当前值会与最后一次提取此行时获取的值进行比较。如果任何值发生改变，则服务器就会知道其他人已更新了此行，并会返回一个错误。如果值是一样的，服务器就执行修改。

本章小结

- 并发调度
- 冲突可串性
- 视图可串性
- 基于锁的并发控制
- 乐观并发控制