

# 数据库系统概论

An Introduction to Database System

## 第12章 并发控制

# 并发控制

## ❖ 多用户数据库系统

允许多个用户同时使用的数据库系统

- 飞机订票系统

- 银行数据库系统

- 网上购物系统

特点：在同一时刻并发运行的事务数可达数百上千个

# 并发控制（续）

## ❖ 多事务执行方式

### （1）事务串行执行

- 每个时刻只有一个事务运行，其他事务必须等到这个事务结束以后方能运行
- 事务在执行过程需要不同的资源，如**CPU**、磁盘、通信
- 串行执行不能充分利用系统资源，不能发挥数据库共享资源的特点

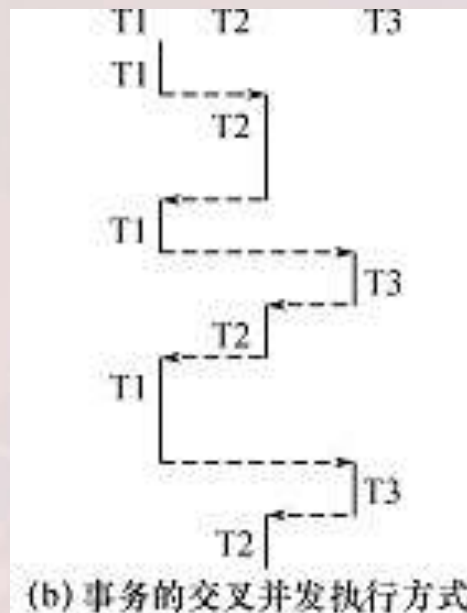


事务的串行执行方式

# 并发控制（续）

## （2）交叉并发方式（Interleaved Concurrency）

- 在单处理机系统中，事务的并行执行是这些并行事务的并行操作轮流交叉运行
- 单处理机系统中的并行事务并没有真正地并行运行，但能够减少处理机的空闲时间，提高系统的效率



# 并发控制（续）

## （3）同时并发方式（**simultaneous concurrency**）

- 多处理机系统中，每个处理机可以运行一个事务，多个处理机可以同时运行多个事务，实现多个事务真正的并行运行
- 最理想的并发方式，但受制于硬件环境
- 更复杂的并发方式机制

❖ 本章讨论的数据库系统并发控制技术是以单处理机系统为基础的

# 并发控制（续）

## ❖ 事务并发执行带来的问题

- 会产生多个事务同时存取同一数据的情况
- 可能会存取和存储不正确的数据，破坏事务隔离性和数据库的一致性

## ❖ 数据库管理系统必须提供并发控制机制

## ❖ 并发控制机制是衡量一个数据库管理系统性能的重要标志之一

$T_1$	$T_2$
① <u>读A=16</u>	
②	<u>读A=16</u>
③ <u>A←A-1</u> <u>写回A=15</u>	
④	<u>A←A-3</u> <u>写回A=13</u>

**T1的修改被T2覆盖了！**

# 第12章 并发控制

## 12.1 并发控制概述

## 12.2 事务的隔离级别

## 12.3 封锁

## 12.4 封锁协议

## 12.5 活锁和死锁

## 12.6 并发调度的可串行性

## 12.7 两段锁协议

## 12.8 封锁的粒度

## \*12.9 其他并发控制机制

## 本章小结

# 12.1 并发控制概述

❖ 事务是并发控制的基本单位

❖ 并发控制机制的任务

- 对并发操作进行正确调度
- 保证事务的隔离性
- 保证数据库的一致性



# 并发控制概述（续）

## 并发操作带来数据的不一致性实例

[例12.1]飞机订票系统中的一个活动序列

- ① 甲售票点(事务 $T_1$ )读出某航班的机票余额 $A$ , 设 $A=16$ ;
  - ② 乙售票点(事务 $T_2$ )读出同一航班的机票余额 $A$ , 也为16;
  - ③ 甲售票点卖出一张机票, 修改余额 $A \leftarrow A-1$ , 所以 $A$ 为15, 把 $A$ 写回数据库;
  - ④ 乙售票点卖出三张机票, 修改余额 $A \leftarrow A-3$ , 所以 $A$ 为13, 把 $A$ 写回数据库
- 结果明明卖出4张机票, 数据库中机票余额只减少3

$T_1$	$T_2$
① <u>读<math>A=16</math></u>	
②	<u>读<math>A=16</math></u>
③ <u><math>A \leftarrow A-1</math></u> <u>写回<math>A=15</math></u>	
④	<u><math>A \leftarrow A-3</math></u> <u>写回<math>A=13</math></u>

**T1的修改被T2覆盖了!**

# 并发控制概述（续）

- ❖ 这种情况称为数据库的不一致性，是由并发操作引起的。
- ❖ 在并发操作情况下，对 $T_1$ 、 $T_2$ 两个事务的操作序列的调度是随机的。
- ❖ 若按上面的调度序列执行， $T_1$ 事务的修改就被丢失。
  - 原因：第④步中 $T_2$ 事务修改A并写回后覆盖了 $T_1$ 事务的修改

# 并发控制概述（续）

## ❖ 并发操作带来的数据不一致性

1. 丢失修改

2. 脏读（**dirty read**）

3. 不可重复读

\*4. 幻读（**phantom row**）

## ❖ 记号

■ **R(x)**: 读数据x

■ **W(x)**: 写数据x

# 1. 丢失修改

❖ 两个事务 $T_1$ 和 $T_2$ 读入同一数据并修改， $T_2$ 的提交结果破坏了 $T_1$ 提交的结果，导致 $T_1$ 的修改被丢失。

$T_1$	$T_2$
① $R(A)=16$	
②	$R(A)=16$
③ $A \leftarrow A-1$	
$W(A)=15$	
④	$A \leftarrow A-3$
	$W(A)=13$



写-写

丢失修改

## 2. 脏读（dirty read）

也称读“脏”数据，指：

- 事务 $T_1$ 修改某一数据，并将其写回磁盘
- 事务 $T_2$ 读取同一数据后， $T_1$ 由于某种原因被撤销
- 这时 $T_1$ 已修改过的数据恢复原值， $T_2$ 读到的数据就与数据库中的数据不一致
- $T_2$ 读到的数据就为“脏”数据，即不正确的数据

# 脏读 ( dirty read ) (续)

例如:

$T_1$	$T_2$
① $R(C)=100$ $C \leftarrow C * 2$ $W(C)=200$	
②	$R(C)=200$
③ ROLLBACK C恢复为100	

读“脏”数据

修改—读

- $T_1$ 将C值修改为200
- $T_2$ 读到C为200
- $T_1$ 由于某种原因撤销，其修改作废，C恢复原值100。
- 这时 $T_2$ 读到的C为200，与数据库内容不一致，就是“脏”数据

### 3. 不可重复读

❖ 不可重复读是指事务 $T_1$ 读取数据后，事务 $T_2$ 执行更新操作，使 $T_1$ 无法再现前一次读取结果。



读-更新

# 不可重复读（续）

例如：

T <sub>1</sub>	T <sub>2</sub>
① R(A)=50 R(B)=100 求和=150	
②	R(B)=100 B←B*2 W(B)=200
③ R(A)=50 R(B)=200 求和=250 (验算不对)	

不可重复读

- T<sub>1</sub>读取B=100进行运算
- T<sub>2</sub>读取同一数据B，对其进行修改后将B=200写回数据库。
- T<sub>1</sub>为了对读取值校对重读B，B已为200，与第一次读取值不一致

读—修改



## \*4. 幻读 ( phantom row )

❖ 幻读也称作幻影 ( phantom row ) 现象, 是指事务 $T_1$ 读取数据后, 事务 $T_2$ 执行插入或删除操作, 使 $T_1$ 无法再现前一次读取结果。

读—删除

(1) 事务 $T_1$ 按一定条件从数据库中读取某些数据记录后, 事务 $T_2$ 删除了其中部分记录, 当 $T_1$ 再次按相同条件读取数据时, 发现某些记录“神秘地”消失了。

读—插入

(2) 事务 $T_1$ 按一定条件从数据库中读取某些数据记录后, 事务 $T_2$ 插入了一些记录, 当 $T_1$ 再次按相同条件读取数据时, 发现多了一些记录。

# 并发控制概述（续）

- ❖ 数据不一致性：由于并发操作破坏了事务的隔离性
- ❖ 并发控制就是要用正确的方式调度并发操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致性
- ❖ 对数据库的应用有时允许某些不一致性，例如有些统计工作涉及数据量很大，读到一些“脏”数据对统计精度没什么影响，可以降低对一致性的要求以减少系统开销

# 并发控制概述（续）

## ❖ 并发控制的主要技术

- 封锁(Locking)

- 时间戳(Timestamp)

- 乐观控制法

- 多版本并发控制(MVCC)

# 第12章 并发控制

12.1 并发控制概述

12.2 事务的隔离级别

12.3 封锁

12.4 封锁协议

12.5 活锁和死锁

12.6 并发调度的可串行性

12.7 两段锁协议

12.8 封锁的粒度

\*12.9 其他并发控制机制

本章小结

## 12.2 事务的隔离级别

- ❖ 为防止数据不一致，需要数据库管理系统对并发操作进行控制
- ❖ 这种控制越严格，事务的隔离性就越强，数据的一致性就越有保障，但系统的效率也会随之下降。
- ❖ **SQL**标准中给出了事务的四类隔离级别，以满足不同应用场景的需求
  - 四类隔离级别由低到高分别是：读未提交、读已提交、可重复读、可串行化

# 1.读未提交

- ❖ “读未提交”是允许一个事务可以读取另一个未提交事务正在修改的数据。它可能出现脏读、不可重复读和幻读的情形。

## 2.读已提交

- ❖ “读已提交”是只允许一个事务读其他事务已提交的数据。显然，“读已提交”可以有效避免脏读，但是它不能保证可重复读和不幻读。

### 3.可重复读

- ❖ “可重复读”是一个事务开始读取数据后，其他事务就不能再对该数据执行**UPDATE**操作了。
- ❖ “可重复读”杜绝了脏读和不可重复读
- ❖ 不能保证不幻读



## 4.可串行化

- ❖ “可串行化”是最高的事务隔离级别
- ❖ 事务执行顺序是可串行化的，可以避免丢失修改、脏读、不可重复读和幻读

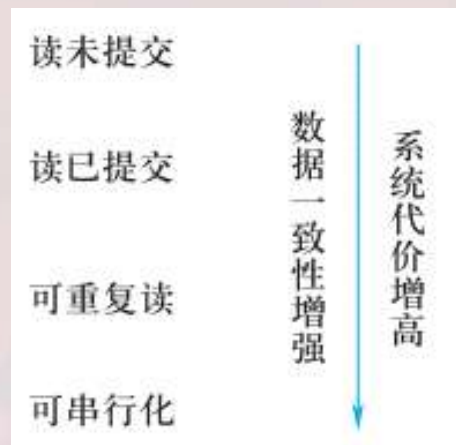
# 事务隔离级别与数据不一致性的关系

表 12.1 事务隔离级别与数据不一致性的关系

事务隔离级别	数据不一致性			
	丢失修改	脏读	不可重复读	* 幻读
读未提交	否	是	是	是
读已提交	否	否	是	是
可重复读	否	否	否	是
可串行化	否	否	否	否

# 事务隔离级别与数据一致性以及系统代价的关系

- 事务隔离级别并不是越高越好
- 应根据应用的特点和需求选择合适的事务隔离级别
- 目前大多数数据库默认的事务隔离级别是“读已提交”，如KingBase、SQL Server、Oracle等。
- MySql的默认级别是“可重复读”



# 第12章 并发控制

12.1 并发控制概述

12.2 事务的隔离级别

12.3 封锁

12.4 封锁协议

12.5 活锁和死锁

12.6 并发调度的可串行性

12.7 两段锁协议

12.8 封锁的粒度

\*12.9 其他并发控制机制



本章小结

## 12.3 封锁

- ❖ 什么是封锁
- ❖ 基本封锁类型
- ❖ 锁的相容矩阵

# 什么是封锁

- ❖ 封锁就是事务T在对某个数据对象（例如表、记录等）操作之前，先向系统发出请求，对其加锁
- ❖ 加锁后事务T就对该数据对象有了一定的控制，在事务T释放它的锁之前，其它的事务不能更新此数据对象。
- ❖ 封锁是实现并发控制的一个非常重要的手段

T <sub>1</sub>	T <sub>2</sub>
①  R(A)=16	
②	R(A)=16
③ A←A-1	
 W(A)=15	
④	A←A-3
	W(A)=13

# 基本封锁类型

❖ 一个事务对某个数据对象加锁后究竟拥有什么样的控制由封锁的类型决定。

## ❖ 基本封锁类型

- 排它锁（**Exclusive Locks**，简记为**X锁**）

- 共享锁（**Share Locks**，简记为**S锁**）

# 排它锁

- ❖ 排它锁又称为写锁，**X锁**
- ❖ 若事务**T**对数据对象**A**加上**X锁**，则只允许**T**读取和修改**A**，其它任何事务都不能再对**A**加任何类型的锁，直到**T**释放**A**上的锁
- ❖ 保证其他事务在**T**释放**A**上的锁之前不能再读取和修改**A**



# 共享锁

- ❖ 共享锁又称为读锁，**S**锁
- ❖ 若事务**T**对数据对象**A**加上**S**锁，则事务**T**可以读**A**但不能修改**A**，其它事务只能再对**A**加**S**锁，而不能加**X**锁，直到**T**释放**A**上的**S**锁
- ❖ 保证其他事务可以读**A**，但在**T**释放**A**上的**S**锁之前不能对**A**做任何修改

# 锁的相容矩阵

$T_1 \backslash T_2$	X	S	—
X	N	N	Y
S	N	Y	Y
—	Y	Y	Y

Y=Yes, 相容的请求  
N=No, 不相容的请求

# 第12章 并发控制

12.1 并发控制概述

12.2 事务的隔离级别

12.3 封锁

12.4 封锁协议

12.5 活锁和死锁

12.6 并发调度的可串行性

12.7 两段锁协议

12.8 封锁的粒度

\*12.9 其他并发控制机制

本章小结

## 12.4 封锁协议

### ❖ 什么是封锁协议

- 在运用**X**锁和**S**锁对数据对象加锁时，需要约定一些规则，这些规则为封锁协议（**Locking Protocol**）。
  - 何时申请**X**锁或**S**锁
  - 持锁时间
  - 何时释放
- 对封锁方式规定不同的规则，就形成了各种不同的封锁协议。

# 封锁协议（续）

## ❖ 三级封锁协议

- 对并发操作的不正确调度可能会带来丢失修改、不可重复读和脏读等数据不一致性问题
- 三级封锁协议分别在不同程度上解决了这些问题，为并发操作的正确调度提供一定的保证。
- 不同级别的封锁协议达到的数据一致性级别是不同的

# 保持数据一致性的常用封锁协议

## ❖ 三级封锁协议

1. 一级封锁协议

2. 二级封锁协议

3. 三级封锁协议

# 1. 一级封锁协议

## ❖ 一级封锁协议

- 事务T在修改数据R之前必须先对其加X锁，直到事务结束才释放。

- 正常结束（**COMMIT**）

- 非正常结束（**ROLLBACK**）

## ❖ 一级封锁协议可防止丢失修改，并保证事务T是可恢复的。

# 使用封锁机制解决丢失修改问题

例：

T <sub>1</sub>	T <sub>2</sub>
① <u>R(A)=16</u>	
②	<u>R(A)=16</u>
③ <u>A←A-1</u> <u>W(A)=15</u>	
④	<u>A←A-3</u> <u>W(A)=13</u>

T <sub>1</sub>	T <sub>2</sub>
① <u>Xlock A</u>	
② R(A)=16	
	<u>Xlock A</u>
③ A←A-1	等待
W(A)=15	等待
<u>Commit</u>	等待
<u>Unlock A</u>	等待
④	获得Xlock A
	<u>R(A)=15</u>
	<u>A←A-3</u>
⑤	<u>W(A)=12</u>
	Commit
没有丢失修改	Unlock A



# 一级封锁协议（续）

- ❖ 在一级封锁协议中，如果仅仅是读数据不对其进行修改，是不需要加锁的，所以它不能保证可重复读和不读“脏”数据。

# 使用一级封锁协议不能解决的问题

$T_1$	$T_2$
① $R(A)=50$ $R(B)=100$ 求和=150	
②	$R(B)=100$ $B \leftarrow B * 2$ $W(B)=200$
③ $R(A)=50$ $R(B)=200$ 求和=250 (验算不对)	

不可重复读

$T_1$	$T_2$
① $R(A)=50$ $R(B)=100$ 求和=150	
②	$Xlock B$ 获得 $R(B)=100$ $B \leftarrow B * 2$ $W(B)=200$ Commit Unlock B
③ $R(A)=50$ $R(B)=200$ 求和=250 (验算不对)	

不可重复读

# 使用一级封锁协议不能解决的问题

T <sub>1</sub>	T <sub>2</sub>
① R(C)=100 C←C*2 W(C)=200	
②	R(C)=200
③ ROLLBACK C恢复为100	

读“脏”数据

T <sub>1</sub>	T <sub>2</sub>
① Xlock C 获得	
② R(C)=100 C←C*2 W(C)=200	
③	R(C)=200
④ Rollback C恢复为100 Unlock C	

读“脏”数据

## 2. 二级封锁协议

### ❖ 二级封锁协议

- 一级封锁协议基础上，增加事务**T**在读取数据**R**之前必须先对其加**S**锁，读完后即可释放**S**锁。

### ❖ 二级封锁协议可以防止丢失修改和读“脏”数据。

# 使用二级封锁协议解决丢失修改问题

例:

$T_1$	$T_2$
① Xlock A	
② R(A)=16	
	Xlock A
③ $A \leftarrow A-1$	等待
$W(A)=15$	等待
Commit	等待
Unlock A	等待
④	获得Xlock A
	R(A)=15
	$A \leftarrow A-3$
⑤	W(A)=12
	Commit
	Unlock A

没有丢失修改

- 事务T1在读A进行修改之前先对A加X锁
- 当T2再请求对A加X锁时被拒绝
- T2只能等待T1释放A上的锁后T2获得对A的X锁
- 这时T2读到的A已经是T1更新过的值15
- T2按此新的A值进行运算, 并将结果值A=12送回到磁盘。避免了丢失T1的更新。

# 使用封锁机制解决读“脏”数据问题

例

T <sub>1</sub>	T <sub>2</sub>
① R(C)=100 C ← C*2 W(C)=200	
②	R(C)=200
③ ROLLBACK C恢复为100	

读“脏”数据

T <sub>1</sub>	T <sub>2</sub>
① Xlock C R(C)=100 C ← C*2 W(C)=200	
②	Slock C
	等待
③ ROLLBACK (C恢复为100) Unlock C	等待
	等待
④	获得Slock C
	R(C)=100
	Commit C
	Unlock C

未读“脏”数据

## 二级封锁协议（续）

- ❖ 在二级封锁协议中，由于读完数据后即可释放S锁，所以它不能保证可重复读。

# 使用二级封锁协议不能解决的问题: 不可重复读

T <sub>1</sub>	T <sub>2</sub>
① R(A)=50 R(B)=100 求和=150	
②	R(B)=100 B←B*2 W(B)=200
③ R(A)=50 R(B)=200 求和=250 (验算不对)	

不可重复读

T <sub>1</sub>	T <sub>2</sub>
① Slock A 获得 读A=50 Unlock A	
② Slock B 获得	
③	Xlock B 等待 等待
④ 读B=100 Unlock B 求和=150	
⑤	获得 读B=100 B←B*2 写回B=200 Commit Unlock B

T <sub>1</sub> (续)	T <sub>2</sub>
⑥ Slock A 获得 读A=50 Unlock A Slock B 获得 读B=200 Unlock B 求和=250 (验算不对)	

不可重复读



### 3. 三级封锁协议

#### ❖ 三级封锁协议

- 一级封锁协议基础上，增加事务T在读取数据R之前必须先对其加S锁，直到事务结束才释放。

#### ❖ 三级封锁协议可防止丢失修改、读脏数据和不可重复读。

# 使用封锁机制解决不可重复读问题

T <sub>1</sub>	T <sub>2</sub>
① <u>Sclock A</u> 获得 <u>读A=50</u> <u>Unlock A</u>	
② <u>Sclock B</u> 获得	
③	<u>Xlock B</u>
④ <u>读B=100</u> <u>Unlock B</u> 求和=150	等待 等待
⑤	获得 <u>读B=100</u> <u>B ← B * 2</u> <u>写回B=200</u> <u>Commit</u> <u>Unlock B</u>

T <sub>1</sub> (续)	T <sub>2</sub>
⑥ <u>Sclock A</u> 获得 <u>读A=50</u> <u>Unlock A</u> <u>Sclock B</u> 获得 <u>读B=200</u> <u>Unlock B</u> 求和=250 <u>(验算不对)</u>	

不可重复读

T <sub>1</sub>	T <sub>2</sub>
① <u>Slock A</u> <u>Slock B</u> <u>R(A)=50</u> <u>R(B)=100</u> <u>求和=150</u>	
②	<u>Xlock B</u> 等待 等待 等待 等待 等待 等待 等待
③ <u>R(A)=50</u> <u>R(B)=100</u> <u>求和=150</u> <u>Commit</u> <u>Unlock A</u> <u>Unlock B</u>	
④	获得XlockB <u>R(B)=100</u> <u>B ← B * 2</u> <u>W(B)=200</u> <u>Commit</u> <u>Unlock B</u>
⑤	

可重复读

# 封锁协议小结

## ❖ 三级协议的主要区别

■ 什么操作需要申请封锁以及何时释放锁（即持锁时间）

## ❖ 不同的封锁协议使事务达到的一致性级别不同

■ 封锁协议级别越高，一致性程度越高

表12.2 不同级别的封锁协议和一致性保证

	X锁		S锁		一致性保证		
	操作结束 释放	事务结束 释放	操作结束 释放	事务结束 释放	不丢失 修改	不读“脏”数 据	可重复 读
一级封锁协议		√			√		
二级封锁协议		√	√		√	√	
三级封锁协议		√		√	√	√	√

# 第12章 并发控制

12.1 并发控制概述

12.2 事务的隔离级别

12.3 封锁

12.4 封锁协议

12.5 活锁和死锁

12.6 并发调度的可串行性

12.7 两段锁协议

12.8 封锁的粒度

\*12.9 其他并发控制机制

本章小结

## 12.5 活锁和死锁

❖ 封锁技术可以有效地解决并行操作的一致性问题，但也带来一些新的问题

- 死锁

- 活锁

# 12.5.1 活锁

举例：

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>
<del>Lock R</del>	•	•	•
•	•	•	•
•	<del>Lock R</del>	<del>Lock R</del>	
•	<del>等待</del>	<del>等待</del>	
Unlock R	等待	•	
	等待	Lock R	
•	等待	•	Lock R
•	等待	•	等待
•	等待	Unlock R	•
	等待	•	Lock R
	等待		•

# 12.5.1 活锁

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>
Lock R	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•
<u>Unlock R</u>	Lock R	Lock R	
	等待	等待	
	等待	等待	
	等待	•	
	等待	<u>Lock R</u>	
•	<u>等待</u>	•	<u>Lock R</u>
•	等待	•	<u>等待</u>
•	等待	Unlock R	•
	等待	•	Lock R
	等待		•

# 12.5.1 活锁

T <sub>1</sub>	T <sub>2</sub>	T <sub>3</sub>	T <sub>4</sub>
Lock R	•	•	•
	•	•	•
•			
•	Lock R		
•	等待	Lock R	
	等待	等待	
Unlock R	等待	•	
	等待	Lock R	
•	等待	•	Lock R
•	等待	•	等待
•	等待	<u>Unlock R</u>	•
	<u>等待</u>	•	<u>Lock R</u>
	等待		•

活锁

T<sub>2</sub>有可能永远等待，这就是活锁的情形



# 活锁（续）

## ❖ 避免活锁：采用先来先服务的策略

- 当多个事务请求封锁同一数据对象时
- 按请求封锁的先后次序对这些事务排队
- 该数据对象上的锁一旦释放，首先批准申请队列中第一个事务获得锁

## 12.5.2 死锁

- ❖ 事务 $T_1$ 封锁了数据 $R_1$
- ❖  $T_2$ 封锁了数据 $R_2$
- ❖  $T_1$ 又请求封锁 $R_2$ ，因 $T_2$ 已封锁了 $R_2$ ，于是 $T_1$ 等待 $T_2$ 释放 $R_2$ 上的锁
- ❖ 接着 $T_2$ 又申请封锁 $R_1$ ，因 $T_1$ 已封锁了 $R_1$ ， $T_2$ 也只能等待 $T_1$ 释放 $R_1$ 上的锁
- ❖ 这样 $T_1$ 在等待 $T_2$ ，而 $T_2$ 又在等待 $T_1$ ， $T_1$ 和 $T_2$ 两个事务永远不能结束，形成死锁

# 死锁（续）

$T_1$	$T_2$
<u>Lock <math>R_1</math></u>	•
	•
	•
•	<u>Lock <math>R_2</math></u>
•	•
•	•
Lock $R_2$	•
等待	
等待	
等待	Lock $R_1$
等待	等待
等待	等待
•	•
•	•
•	•

# 死锁（续）

<b>T<sub>1</sub></b>	<b>T<sub>2</sub></b>
Lock R <sub>1</sub>	•
	•
	•
•	Lock R <sub>2</sub>
•	•
•	•
<u>Lock R<sub>2</sub></u>	•
<u>等待</u>	
<u>等待</u>	
等待	
等待	<u>Lock R<sub>1</sub></u>
等待	<u>等待</u>
	等待
•	•
•	•
•	•

死锁

# 解决死锁的方法

## 两类方法

1. 死锁的预防
2. 死锁的诊断与解除

# 1. 死锁的预防

- ❖ 产生死锁的原因是两个或多个事务都已封锁了一些数据对象，然后又都请求对已为其他事务封锁的数据对象加锁，从而出现死等待。
- ❖ 预防死锁的发生就是要破坏产生死锁的条件

# 死锁的预防（续）

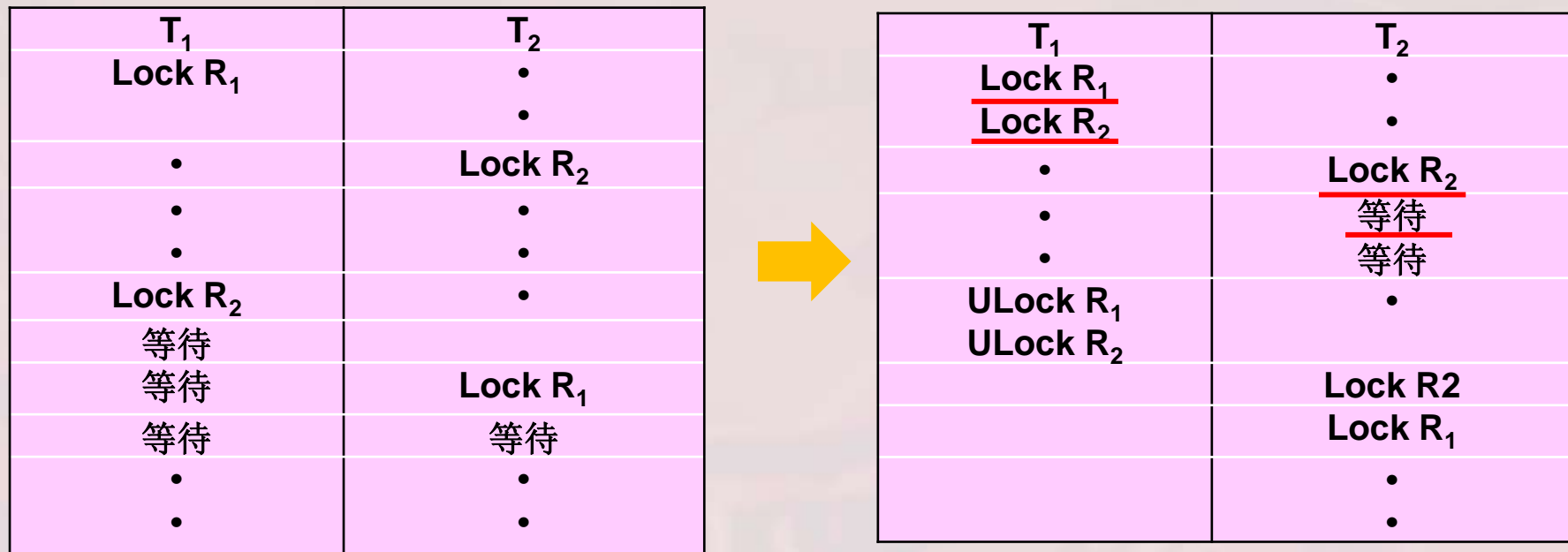
## 预防死锁的方法

**（1）** 一次封锁法

**（2）** 顺序封锁法

# (1) 一次封锁法

❖ 要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行





# (1) 一次封锁法

❖ 要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行

T <sub>1</sub>	T <sub>2</sub>
Lock R <sub>1</sub>	•
•	•
•	Lock R <sub>2</sub>
•	•
Lock R <sub>2</sub>	•
等待	
等待	Lock R <sub>1</sub>
等待	等待
•	•
•	•



T <sub>1</sub>	T <sub>2</sub>
Lock R <sub>1</sub>	•
Lock R <sub>2</sub>	•
•	Lock R <sub>2</sub>
•	等待
•	等待
<u>ULock R<sub>1</sub></u>	•
<u>ULock R<sub>2</sub></u>	
	<u>Lock R<sub>2</sub></u>
	<u>Lock R<sub>1</sub></u>
	•
	•

# 一次封锁法存在的问题

- ❖ 过早加锁，降低系统并发度
- ❖ 难于事先精确确定封锁对象
  - 数据库中数据是不断变化的，原来不要求封锁的数据，在执行过程中可能会变成封锁对象，所以很难事先精确地确定每个事务所要封锁的数据对象。
  - 可能解决方法：扩大封锁范围，如：将事务在执行过程中可能要封锁的数据对象全部加锁，这就进一步降低了并发度。

## (2) 顺序封锁法

❖ 顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实行封锁。

❖ 顺序封锁法存在的问题

### ■ 维护成本

数据库系统中封锁的数据对象极多，并且随数据的插入、删除等操作而不断地变化，要维护这样的资源的封锁顺序非常困难，**成本很高**。

### ■ 难以实现

事务的封锁请求可以随着事务的执行而动态地决定，很难事先确定每一个事务要封锁哪些对象，因此也就**很难按规定的顺序去施加封锁**

# 死锁的预防（续）

## ❖ 结论

- 在操作系统中广为采用的预防死锁的策略并不太适合数据库的特点
- 数据库管理系统在解决死锁的问题上更普遍采用的是诊断并解除死锁的方法

## 2. 死锁的诊断与解除

### ❖ 死锁的诊断

(1) 超时法

(2) 等待图法

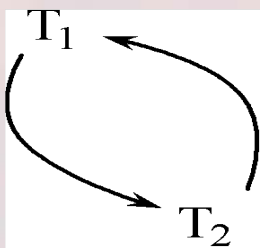
# (1) 超时法

- ❖ 如果一个事务的等待时间超过了规定的时限，就认为发生了死锁
- ❖ 优点：实现简单
- ❖ 缺点
  - 有可能误判死锁
  - 时限若设置得太长，死锁发生后不能及时发现

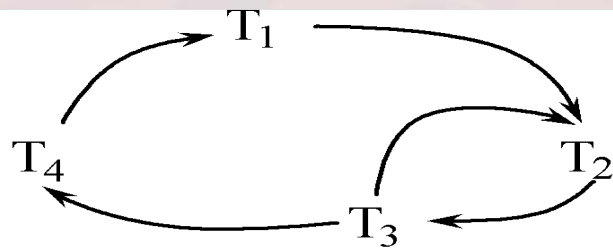
## (2) 等待图法

❖ 用事务等待图动态反映所有事务的等待情况

- 事务等待图是一个有向图  $G=(T, U)$
- $T$  为结点的集合，每个结点表示正运行的事务
- $U$  为边的集合，每条边表示事务等待的情况
- 若  $T_1$  等待  $T_2$ ，则  $T_1, T_2$  之间划一条有向边，从  $T_1$  指向  $T_2$



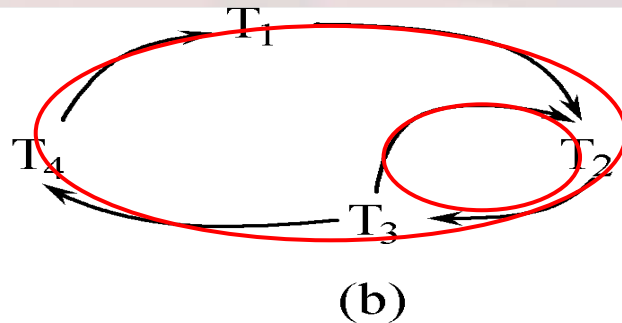
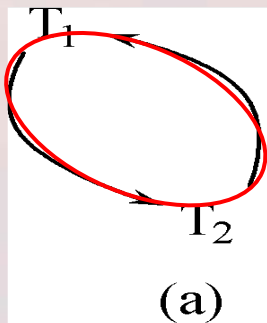
(a)



(b)

## 等待图法（续）

❖ 并发控制子系统周期性地（比如每隔数秒）生成事务等待图，检测事务。如果发现图中存在回路，则表示系统中出现了死锁。

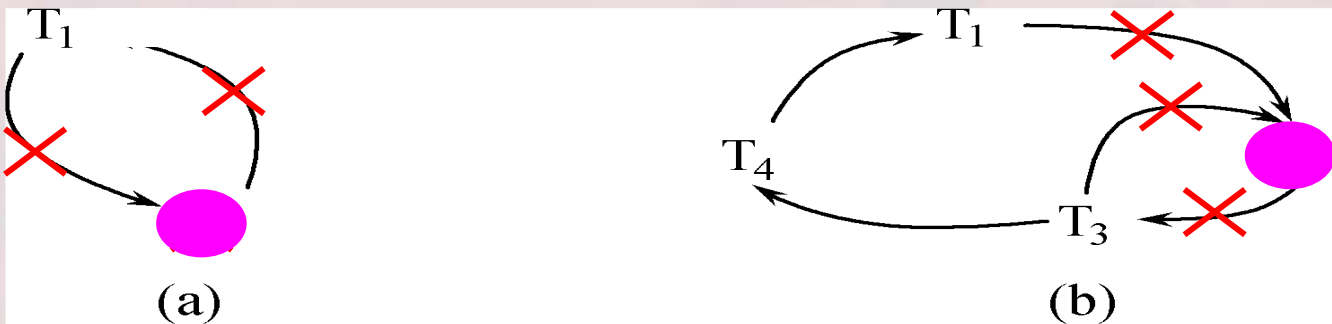




# 死锁的诊断与解除（续）

## ❖ 解除死锁

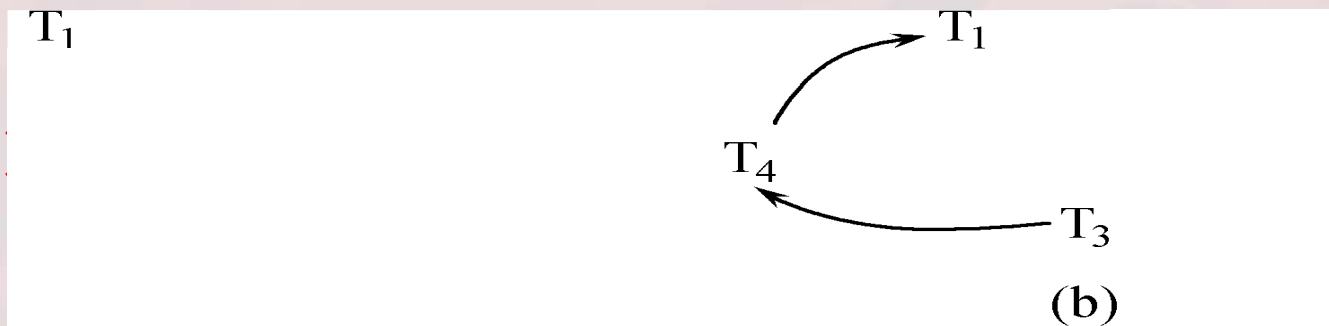
- 选择一个处理死锁代价最小的事务，将其撤消
- 释放此事务持有的所有的锁，使其它事务能继续运行下去



# 死锁的诊断与解除（续）

## ❖ 解除死锁

- 选择一个处理死锁代价最小的事务，将其撤消
- 释放此事务持有的所有的锁，使其它事务能继续运行下去



# 第12章 并发控制

12.1 并发控制概述

12.2 事务的隔离级别

12.3 封锁

12.4 封锁协议

12.5 活锁和死锁

12.6 并发调度的可串行性

12.7 两段锁协议

12.8 封锁的粒度

\*12.9 其他并发控制机制

本章小结

## 12.6 并发调度的可串行性

- ❖ 数据库管理系统对并发事务不同的调度可能会产生不同的结果
- ❖ 串行调度是正确的
- ❖ 执行结果等价于串行调度的调度也是正确的，称为可串行化调度

# 12.6 并发调度的可串行性

## 12.6.1 可串行化调度

## 12.6.2 冲突可串行化调度

## 12.6.1 可串行化调度

### ❖ 可串行化(**Serializable**)调度

- 多个事务的并发执行是正确的，当且仅当其结果与按某一次序串行地执行这些事务时的结果相同

### ❖ 可串行性(**Serializability**)

- 是并发事务正确调度的准则
- 一个给定的并发调度，当且仅当它是可串行化的，才认为是正确调度

# 可串行化调度（续）

**[例]** 现在有两个事务，分别包含下列操作：

■ 事务T1：读B；  $A=B+1$ ；写回A

■ 事务T2：读A；  $B=A+1$ ；写回B

现给出对这两个事务不同的调度策略

# 串行调度,正确的调度

$T_1$	$T_2$
<u>Slock B</u>	
<u><math>Y=R(B)=2</math></u>	
<u>Unlock B</u>	
<u>Xlock A</u>	
<u><math>A=Y+1=3</math></u>	
<u>W(A)</u>	
<u>Unlock A</u>	
	<u>Slock A</u>
	<u><math>X=R(A)=3</math></u>
	<u>Unlock A</u>
	<u>Xlock B</u>
	<u><math>B=X+1=4</math></u>
	<u>W(B)</u>
	<u>Unlock B</u>

- 假设A、B的初值均为2。
- 按 $T_1 \rightarrow T_2$ 次序执行结果为  
 **$A=3, B=4$**
- 串行调度策略, 正确的调度



# 串行调度,正确的调度

$T_1$	$T_2$
	<u>Slock A</u>
	<u><math>X=R(A)=2</math></u>
	<u>Unlock A</u>
	<u>Xlock B</u>
	<u><math>B=X+1=3</math></u>
	<u><math>W(B)</math></u>
	<u>Unlock B</u>
<u>Slock B</u>	
<u><math>Y=R(B)=3</math></u>	
<u>Unlock B</u>	
<u>Xlock A</u>	
<u><math>A=Y+1=4</math></u>	
<u><math>W(A)</math></u>	
<u>Unlock A</u>	

- 假设A、B的初值均为2。
- $T_2 \rightarrow T_1$ 次序执行结果为  
 **$B=3, A=4$**
- 串行调度策略, 正确的调度

# 不可串行化调度，错误的调度

$T_1$	$T_2$
<u>Slock B</u>	
<u><math>Y=R(B)=2</math></u>	
	<u>Slock A</u>
	<u><math>X=R(A)=2</math></u>
<u>Unlock B</u>	
	<u>Unlock A</u>
<u>Xlock A</u>	
<u><math>A=Y+1=3</math></u>	
<u><math>W(A)</math></u>	
	<u>Xlock B</u>
	<u><math>B=X+1=3</math></u>
	<u><math>W(B)</math></u>
<u>Unlock A</u>	
	<u>Unlock B</u>

- 执行结果 **A=3, B=3**,  
与(a)、(b)的结果都不同
- 是错误的调度

# 可串行化调度，正确的调度

$T_1$	$T_2$
<u>Slock B</u>	
<u><math>Y=R(B)=2</math></u>	
<u>Unlock B</u>	
<u>Xlock A</u>	
	<u>Slock A</u>
<u><math>A=Y+1=3</math></u>	<u>等待</u>
<u><math>W(A)</math></u>	<u>等待</u>
<u>Unlock A</u>	<u>等待</u>
	<u><math>X=R(A)=3</math></u>
	<u>Unlock A</u>
	<u>Xlock B</u>
	<u><math>B=X+1=4</math></u>
	<u><math>W(B)</math></u>
	<u>Unlock B</u>

- 执行结果 **A=3, B=4**,  
与第一种串行调度的  
执行结果相同
- 是正确的调度

# 四种调度

T <sub>1</sub>	T <sub>2</sub>
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
A=Y+1=3	
W(A)	
Unlock A	
	Slock A
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

串行调度

T <sub>1</sub>	T <sub>2</sub>
	Slock A
	X=R(A)=2
	Unlock A
	Xlock B
	B=X+1=3
	W(B)
	Unlock B
Slock B	
Y=R(B)=3	
Unlock B	
Xlock A	
A=Y+1=4	
W(A)	
Unlock A	

串行调度

T <sub>1</sub>	T <sub>2</sub>
Slock B	
Y=R(B)=2	
	Slock A
	X=R(A)=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1=3	
W(A)	
	Xlock B
	B=X+1=3
	W(B)
Unlock A	
	Unlock B

不可串行化的调度

T <sub>1</sub>	T <sub>2</sub>
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1=3	等待
W(A)	等待
Unlock A	等待
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

可串行化的调度

# 12.6 并发调度的可串行性

## 12.6.1 可串行化调度

## 12.6.2 冲突可串行化调度

## 12.6.2 冲突可串行化调度

### ❖ 冲突可串行化

#### ■ 一个比可串行化更严格的条件

❖ 冲突操作：指不同的事务对同一数据的读写操作和写写操作：

$R_i(x)$ 与 $W_j(x)$       /\*事务 $T_i$ 读 $x$ ,  $T_j$ 写 $x$ , 其中 $i \neq j$ \*/

$W_i(x)$ 与 $W_j(x)$       /\*事务 $T_i$ 写 $x$ ,  $T_j$ 写 $x$ , 其中 $i \neq j$ \*/

涉及同一个数据库元素, 并且至少有一个是写操作

# 冲突

## ❖ 不冲突操作

- $r_i(X); r_j(X)$  读
- $r_i(X); w_j(Y)$ ,  $X \neq Y$
- $w_i(X); r_j(Y)$ ,  $X \neq Y$
- $w_i(X); w_j(Y)$ ,  $X \neq Y$

# 冲突

❖ 不能交换（**Swap**）的动作：

■ 同一事务的两个操作

■ 不同事务的冲突操作

•  $R_i(x)$ 与 $W_j(x)$

•  $W_i(x)$ 与 $W_j(x)$



# 冲突可串行化

- ❖ 一个调度 $S_c$ 在保证冲突操作的次序不变的情况下，通过交换两个事务不冲突操作的次序得到另一个调度 $S_c'$ ，如果 $S_c'$ 是串行的，称调度 $S_c$ 是冲突可串行化的调度
- ❖ 若一个调度是冲突可串行化，则一定是可串行化的调度
- ❖ 可用这种方法判断一个调度是否是冲突可串行化的

# 冲突可串行化（续）

[例12.3] 今有调度

$Sc_1 = R_1(A)W_1(A)R_2(A)W_2(A)R_1(B)W_1(B)R_2(B)W_2(B)$

$Sc_2 = R_1(A)W_1(A)R_1(B)W_1(B)R_2(A)W_2(A)R_2(B)W_2(B)$

$T_1$   $T_2$

$Sc_2$ 等价于一个串行调度 $T_1, T_2$ ，所以 $Sc_1$ 冲突可串行化的调度

# 冲突可串行化调度

- ❖ 冲突可串行化调度是可串行化调度的充分条件，不是必要条件。还有不满足冲突可串行化条件的可串行化调度。

[例12.4]有3个事务

$$T_1=W_1(Y)W_1(X), \quad T_2=W_2(Y)W_2(X), \quad T_3=W_3(X)$$

- 调度 $L_1=W_1(Y)W_1(X)W_2(Y)W_2(X)W_3(X)$ 是一个串行调度。
- 调度 $L_2=W_1(Y)W_2(Y)W_2(X)W_1(X)W_3(X)$ 不满足冲突可串行化。

但是调度 $L_2$ 是可串行化的，因为 $L_2$ 执行的结果与调度 $L_1$ 相同，Y的值都等于 $T_2$ 的值，X的值都等于 $T_3$ 的值

# 小结

❖ 可串行化调度

❖ 冲突可串行化调度

❖ 可串行化调度与冲突可串行化调度之间的关系

# 第12章 并发控制

12.1 并发控制概述

12.2 事务的隔离级别

12.3 封锁

12.4 封锁协议

12.5 活锁和死锁

12.6 并发调度的可串行性

12.7 两段锁协议

12.8 封锁的粒度

\*12.9 其他并发控制机制

本章小结

## 12.7 两段锁协议

❖ 数据库管理系统普遍采用两段锁协议（**two-phase lock, 2PL**）的方法实现并发调度的可串行性，从而保证调度的正确性

### ❖ 两段锁协议

指所有事务必须分两个阶段对数据项加锁和解锁

- 在对任何数据进行读、写操作之前，事务首先要获得对该数据的封锁
- 在释放一个封锁之后，事务不再申请和获得任何其他封锁

# 两段锁协议（续）

## ❖ “两段” 锁的含义

事务分为两个阶段

### ■ 第一阶段是获得封锁，也称为扩展阶段

- 事务可以申请获得任何数据项上的任何类型的锁，但是不能释放任何锁

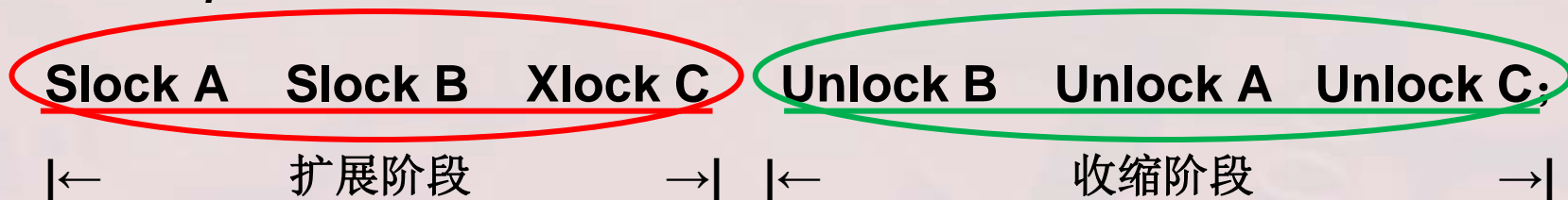
### ■ 第二阶段是释放封锁，也称为收缩阶段

- 事务可以释放任何数据项上的任何类型的锁，但是不能再申请任何锁

# 两段锁协议（续）

例：

事务 $T_i$ 遵守两段锁协议，其封锁序列是：

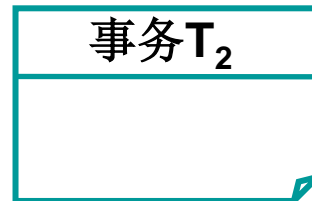
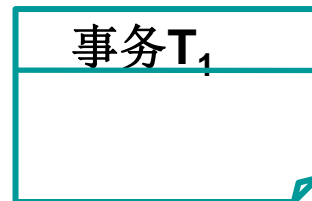
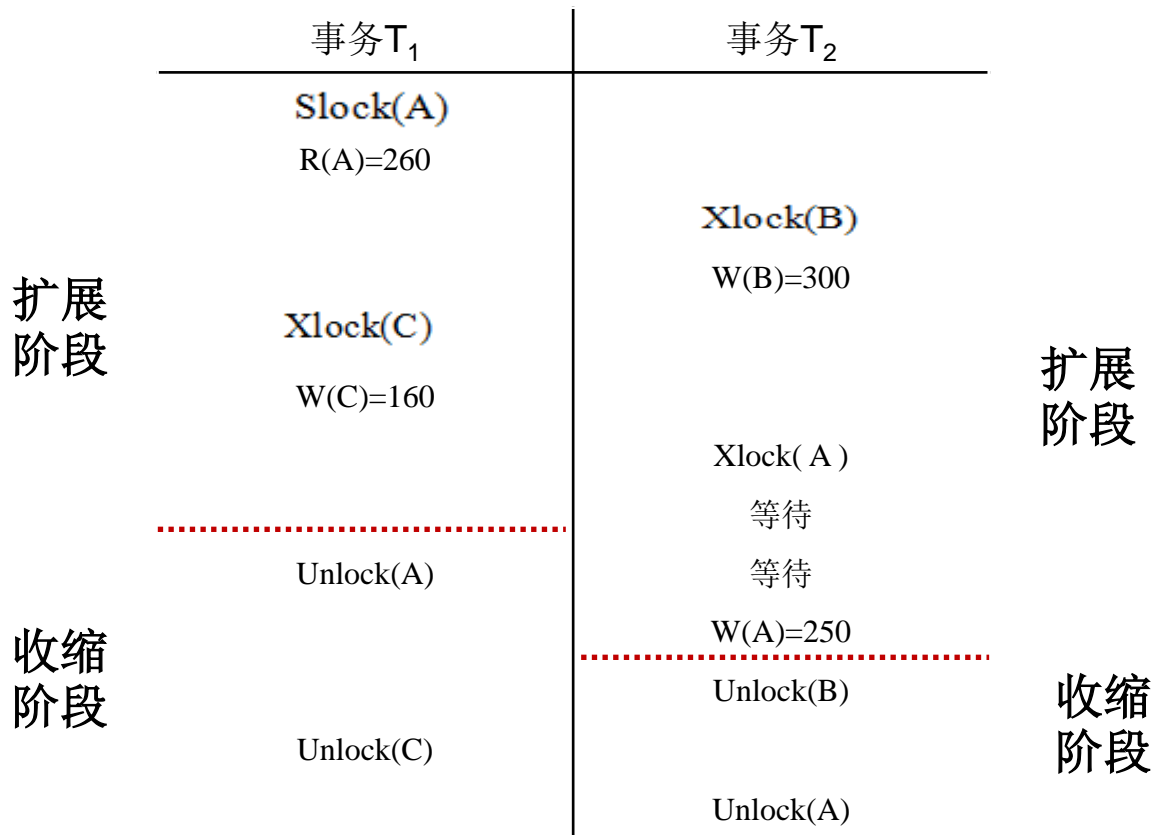


事务 $T_j$ 不遵守两段锁协议，其封锁序列是：





# 两段锁协议（续）



- 遵守两段锁协议，是一个可串行化调度。
- 如何验证？

# 两段锁协议（续）

事务T <sub>1</sub>	事务T <sub>2</sub>
Slock(A)	
R(A)=260	
	Xlock(B)
	W(B)=300
Xlock(C)	
W(C)=160	
	Xlock(A)
	等待
Unlock(A)	等待
	W(A)=250
	Unlock(B)
Unlock(C)	
	Unlock(A)

$L_1 = R_1(A) W_2(B) W_1(C) W_2(A)$



$L_S = \underline{R_1(A) W_1(C)} \underline{W_2(B) W_2(A)}$

# 两段锁协议（续）

- ❖ 事务遵守两段锁协议是可串行化调度的充分条件，而不是必要条件。
- ❖ 若并发事务都遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的
- ❖ 若并发事务的一个调度是可串行化的，不一定所有事务都符合两段锁协议

# 两段锁协议（续）

## ❖ 两段锁协议与防止死锁的一次封锁法的异同

- 一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行，因此一次封锁法遵守两段锁协议
- 但是两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议的事务可能发  
生死锁

# 两段锁协议（续）

[例] 遵守两段锁协议的事务可能发生死锁

事务 $T_1$	事务 $T_2$
<u>Slock B</u>	
<u>R(B)=2</u>	
	<u>Slock A</u>
	<u>R(A)=2</u>
<u>Xlock A</u>	
<u>等待</u>	<u>Xlock B</u>
<u>等待</u>	<u>等待</u>

死锁

# 第12章 并发控制

12.1 并发控制概述

12.2 事务的隔离级别

12.3 封锁

12.4 封锁协议

12.5 活锁和死锁

12.6 并发调度的可串行性

12.7 两段锁协议

12.8 封锁的粒度

\*12.9 其他并发控制机制

本章小结

## 12.8 封锁粒度

❖ 封锁对象的大小称为封锁粒度(**Granularity**)

❖ 封锁的对象:逻辑单元, 物理单元

例: 关系数据库中的封锁对象

■ 逻辑单元: 属性值、属性值的集合、元组、关系、索引项、整个索引、整个数据库等

■ 物理单元: 页(数据页或索引页)、物理记录等

# 选择封锁粒度原则

❖ 封锁粒度与系统的并发度和并发控制的开销密切相关。

- 封锁的粒度越大，数据库所能够封锁的数据单元就越少，并发度就越小，系统开销也越小；
- 封锁的粒度越小，并发度较高，但系统开销也就越大



# 选择封锁粒度的原则（续）

例

- ❖ 若封锁粒度是数据页，事务 $T_1$ 需要修改元组 $L_1$ ，则 $T_1$ 必须对包含 $L_1$ 的整个数据页 $A$ 加锁。如果 $T_1$ 对 $A$ 加锁后事务 $T_2$ 要修改 $A$ 中元组 $L_2$ ，则 $T_2$ 被迫等待，直到 $T_1$ 释放 $A$
- ❖ 如果封锁粒度是元组，则 $T_1$ 和 $T_2$ 可以分别对 $L_1$ 和 $L_2$ 加锁，不需要互相等待，提高了系统的并行度
- ❖ 又如，事务 $T$ 需要读取整个表，若封锁粒度是元组， $T$ 必须对表中的每一个元组加锁，开销极大

# 选择封锁粒度的原则（续）

## ❖ 多粒度封锁(Multiple Granularity Locking)

在一个系统中同时支持多种封锁粒度供不同的事务选择

## ❖ 选择封锁粒度

同时考虑封锁开销和并发度两个因素, 适当选择封锁粒度

- 需要处理大量元组的用户事务：以关系为封锁单元
- 需要处理多个关系的大量元组的用户事务：以数据库为封锁单位
- 只处理少量元组的用户事务：以元组为封锁单位

# 12.8 封锁的粒度

## 12.8.1 多粒度封锁

## 12.8.2 意向锁

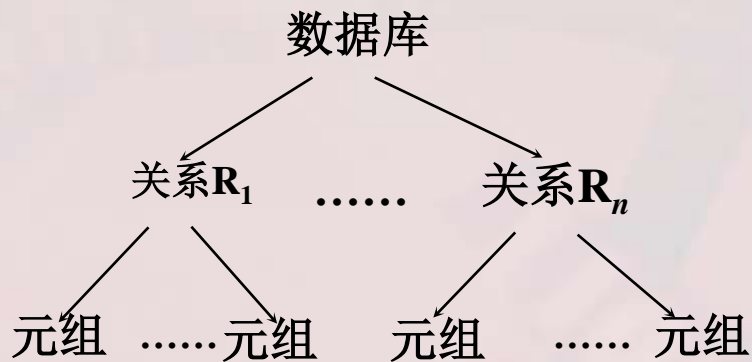
## 12.8.1 多粒度封锁

### ❖ 多粒度树

- 以树形结构来表示多级封锁粒度
- 根结点是整个数据库，表示最大的数据粒度
- 叶结点表示最小的数据粒度

# 多粒度封锁（续）

例：三级粒度树。根结点为数据库，数据库的子结点为关系，关系的子结点为元组。



三级粒度树

# 多粒度封锁协议

- ❖ 允许多粒度树中的每个结点被独立地加锁
- ❖ 对一个结点加锁意味着这个结点的所有后裔结点也被加以同样类型的锁
- ❖ 在多粒度封锁中一个数据对象可能以两种方式封锁：显式封锁和隐式封锁

# 显式封锁和隐式封锁

- ❖ 显式封锁: 直接加到数据对象上的封锁
- ❖ 隐式封锁: 是该数据对象没有独立加锁, 是由于其上级结点加锁而使该数据对象加上了锁
- ❖ 显式封锁和隐式封锁的效果是一样的

# 显式封锁和隐式封锁（续）

## ❖ 系统检查封锁冲突时

- 要检查显式封锁和隐式封锁

## ❖ 例如事务T要对关系 $R_1$ 加X锁

- 系统必须搜索其上级结点数据库、关系 $R_1$
- 还要搜索 $R_1$ 的下级结点，即 $R_1$ 中的每一个元组
- 如果其中某一个数据对象已经加了不相容锁，则T必须等待



# 显式封锁和隐式封锁（续）

## ❖ 对某个数据对象加锁，系统要检查

### ■ 该数据对象

- 有无显式封锁与之冲突

### ■ 所有上级结点

- 检查本事务的显式封锁是否与该数据对象上的隐式封锁冲突：（由上级结点已加的封锁造成的）

### ■ 所有下级结点

- 看上面的显式封锁是否与本事务的隐式封锁（将加到下级结点的封锁）冲突

# 12.8 封锁的粒度

## 12.8.1 多粒度封锁

## 12.8.2 意向锁

## 12.8.2 意向锁

- ❖ 引进意向锁（**intention lock**）目的
  - 提高对某个数据对象加锁时系统的检查效率
- ❖ 如果对一个结点加意向锁，则说明该结点的下层结点正在被加锁
- ❖ 对任一结点加锁，必须先对它的上层结点加意向锁
  - 例如，对任一元组加锁时，必须先对它所在的数据库和关系加意向锁

# 常用意向锁

- ❖ 意向共享型锁(intent shared lock, 简称IS锁)
- ❖ 意向排它型锁(intent exclusive lock, 简称IX锁)
- ❖ 共享意向排它型锁(shared and intention exclusive lock, 简称SIX锁)

# 意向锁（续）

## ❖ IS锁

- 如果对一个数据对象加**IS**锁，表示它的后裔结点拟（意向）加**S**锁。

例如：事务 $T_1$ 要对 $R_1$ 中某个元组加**S**锁，则要首先对关系 $R_1$ 和数据库加**IS**锁

# 意向锁（续）

## ❖ IX锁

- 如果对一个数据对象加IX锁，表示它的后裔结点拟（意向）加X锁。

例如：事务 $T_1$ 要对 $R_1$ 中某个元组加X锁，则要首先对关系 $R_1$ 和数据库加IX锁

# 意向锁（续）

## ❖ SIX锁

- 如果对一个数据对象加**SIX**锁，表示对它加**S**锁，再加**IX**锁，即 **SIX = S + IX**

例：对某个表加**SIX**锁，则表示该事务要读整个表（所以要对该表加**S**锁），同时会更新个别元组（所以要对该表加**IX**锁）

# 意向锁（续）

## 意向锁的相容矩阵

T <sub>1</sub>	T <sub>2</sub>					
	S	X	IS	IX	SIX	—
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
—	Y	Y	Y	Y	Y	Y

注：Y=Yes, 表示相容的请求；N=No, 表示不相容的请求

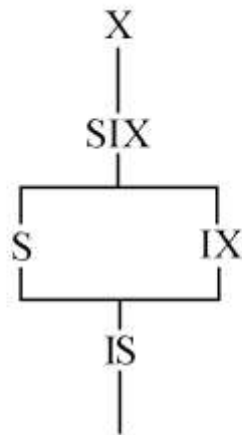
(a) 封锁类型的相容矩阵



# 意向锁（续）

## ❖ 锁的强度

- 锁的强度是指它对其他锁的排斥程度
- 一个事务在申请封锁时以强锁代替弱锁是安全的，反之则不然



(b) 锁的强度的偏序关系

# 意向锁（续）

## ❖ 具有意向锁的多粒度封锁方法

- 申请封锁时应该按自上而下的次序进行
- 释放封锁时则应该按自下而上的次序进行

例如：事务 $T_1$ 要对关系 $R_1$ 加S锁

- 要首先对数据库加IS锁
- 检查数据库是否已加了不相容的锁(X锁)
- 然后检查 $R_1$ 是否已加了不相容的锁(X或IX锁)
- 不再需要搜索和检查 $R_1$ 中的元组是否加了不相容的锁(X锁)

# 意向锁（续）

## ❖ 具有意向锁的多粒度封锁方法

- 提高了系统的并发度
- 减少了加锁和解锁的开销
- 在实际的数据库管理系统产品中得到广泛应用

# 第12章 并发控制

**12.1 并发控制概述**

**12.2 事务的隔离级别**

**12.3 封锁**

**12.4 封锁协议**

**12.5 活锁和死锁**

**12.6 并发调度的可串行性**

**12.7 两段锁协议**

**12.8 封锁的粒度**

**\*12.9 其他并发控制机制**

**本章小结**

# 本章小结

❖ 数据库的并发控制以事务为单位

❖ 并发操作带来的数据不一致性

1. 丢失修改 (**Lost Update**)

2. 不可重复读 (**Non-repeatable Read**)

3. 读“脏”数据 (**Dirty Read**)

# 本章小结（续）

- ❖ 数据库的并发控制通常使用封锁机制
  - 基本封锁（X锁和S锁）
- ❖ 活锁和死锁
  - 活锁
  - 死锁（预防，检测）
- ❖ 解决数据不一致的并发控制协议：三级封锁协议

# 本章小结（续）

## ❖ 并发事务调度的正确性

### ■ 可串行性

- 并发操作的正确性则通常由两段锁协议来保证。
- 两段锁协议是可串行化调度的充分条件，但不是必要条件

### ■ 冲突可串行性

### ■ 两段锁协议—实现并发调度的可串行性

## ❖ 封锁的粒度

# 本章小结（续）

## ❖ 本章目标

- 了解数据库并发控制技术的必要性，掌握并发控制的相关技术。

## ❖ 本章重点

- 牢固掌握并发操作可能产生数据不一致性的情况；基本封锁方法，相关的相容控制矩阵；
- 举一反三：三级封锁协议与数据一致性的关系；并发调度的可串行性概念；封锁粒度与系统并发度和并发控制的开销之间的关系。