

数据库系统

第11章 并发控制

胡 敏

合肥工业大学计算机与信息学院

情感计算与先进智能机器安徽省重点实验室

jsjxhumin@hfut.edu.cn

QQ:495109389

第11章 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

★ 飞机订票数据库系统

★ 银行数据库系统

■ 特点：在同一时刻并发运行的事务数可达数百上千个

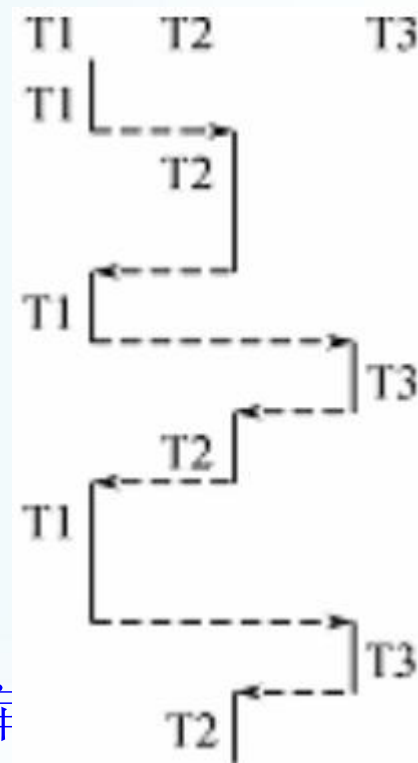
■ 不同的多事务执行方式

- (1) 事务串行执行
- (2) 交叉并发方式
- (3) 同时并发方式：多处理机系统

■ 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，破坏事务一致性和数据库

■ 本章：单处理机



11 并发控制

11.1 并发控制概述

11.2 封锁

11.3 封锁协议

11.4 活锁和死锁

11.5 并发调度的可串行性

11.6 两段锁协议

11.7 封锁的粒度

*11.8 其他并发控制机制

11.9 小结



11.1 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

例如：对于一个转帐活动：A帐户转帐给B帐户n元钱，
这个活动包含两个动作：

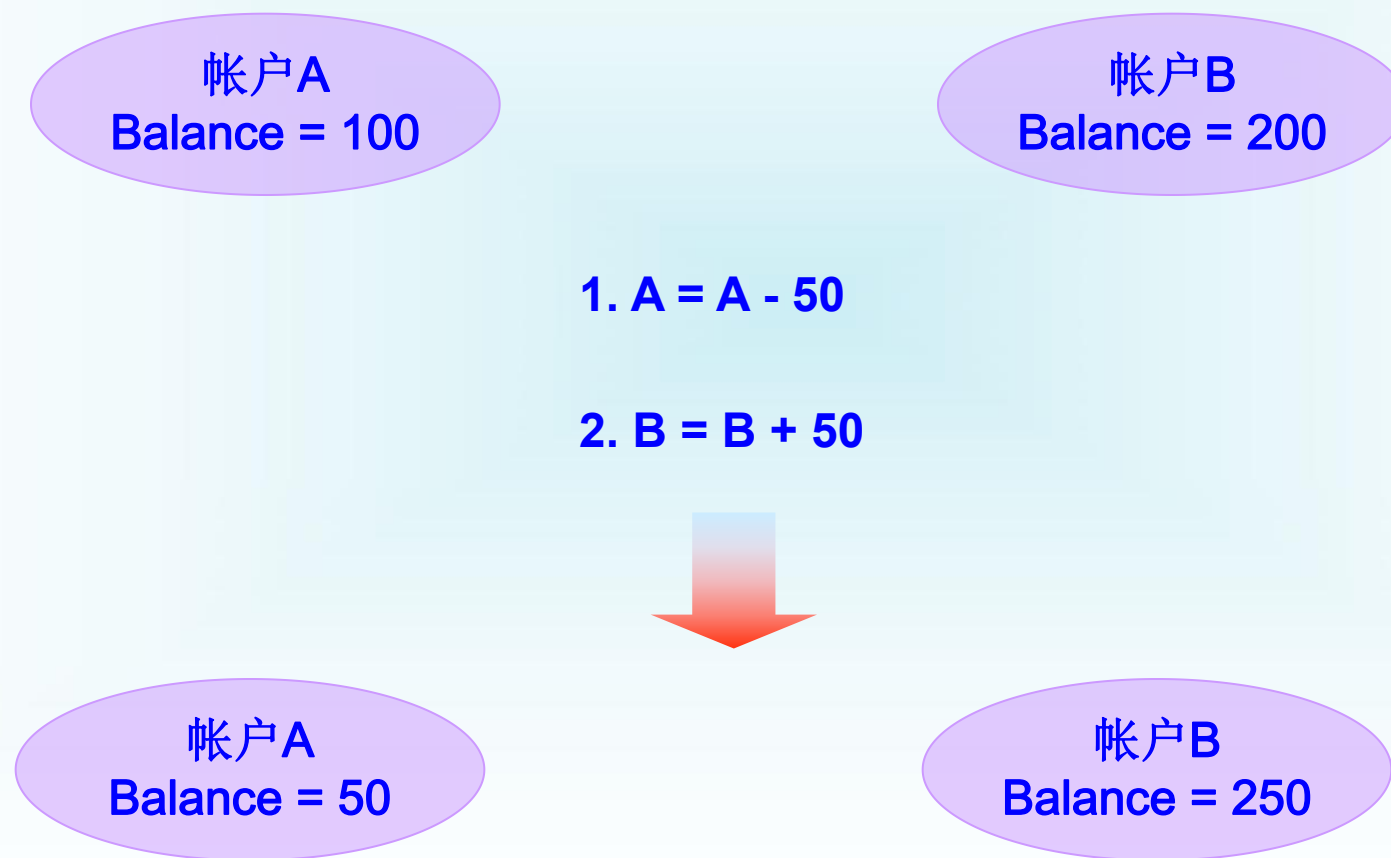
第一个动作：A帐户 - n

第二个动作：B帐户 + n



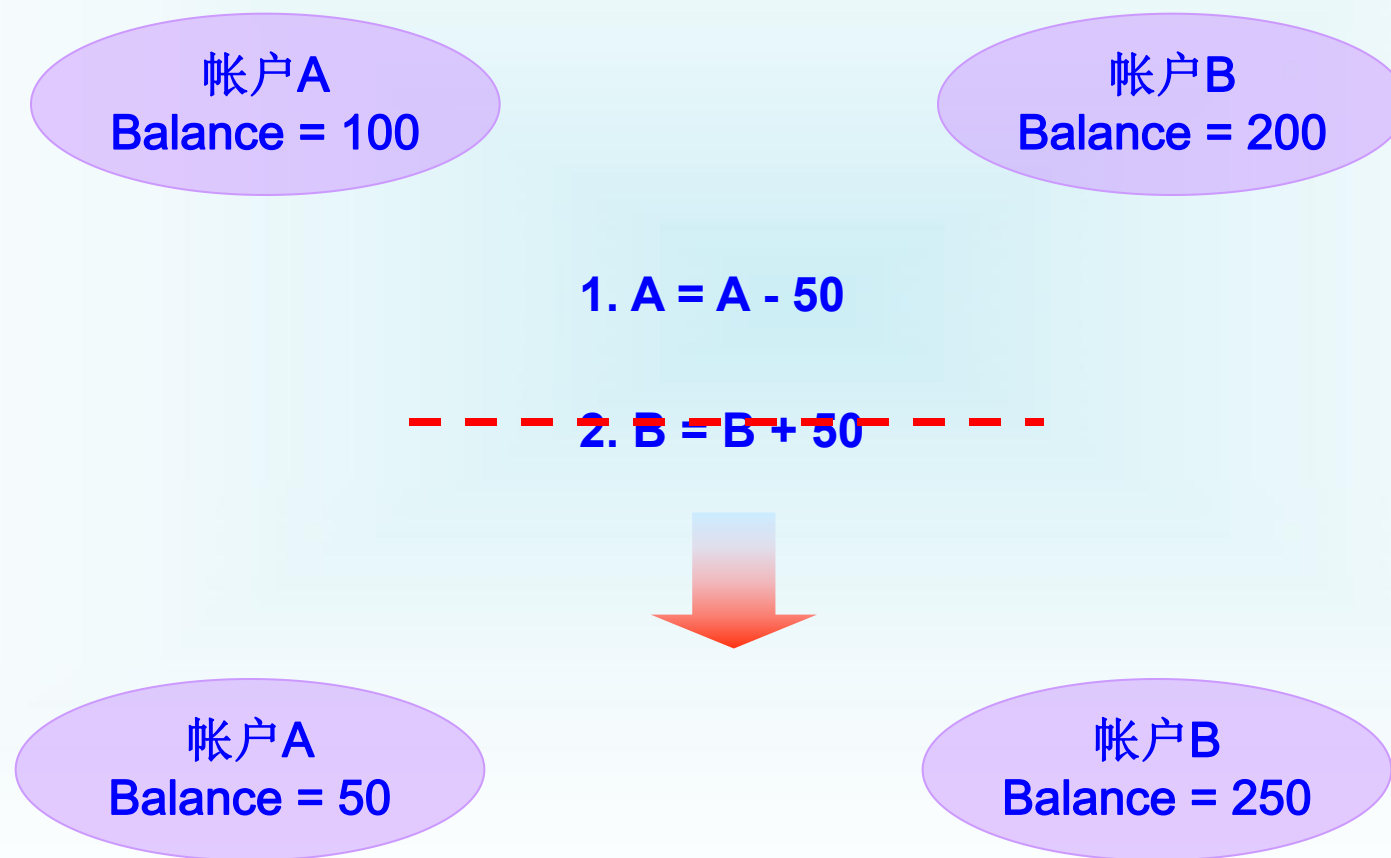
为什么需要事务

- 防止数据库中数据的不一致性。



为什么需要事务

- 防止数据库中数据的不一致性。



为什么需要事务

■ 并发控制以事务为单位进行的

例: **BEGIN TRANSACTION**

UPDATE 支付表 SET 帐户总额 = 帐户总额 - n WHERE 帐户名 = 'A'

UPDATE 支付表 SET 帐户总额 = 帐户总额 + n WHERE 帐户名 = 'B'

COMMIT



11.1 并发控制概述

■ 实例分析

[例] 飞机订票系统中的一个活动序列

- ① 甲售票点(甲事务)读出某航班的机票余额A, 设A=16;
- ② 乙售票点(乙事务)读出同一航班的机票余额A, 也为16;
- ③ 甲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;
- ④ 乙售票点卖出四张机票, 修改余额 $A \leftarrow A-4$, 所以A为12, 把A写回数据库。

■ 并发操作带来的数据不一致性

- 丢失修改
- 不可重复读
- 读“脏”数据

T1	T2
① 读A=16	读A=16
②	
③ $A \leftarrow A-3$ 写回A=15	$A \leftarrow A-4$ 写回A=12
④	

T1的修改被T2覆盖了!

1. 丢失修改

■ 记号

$R(x)$: 读数据 x $W(x)$: 写数据 x

- 两个事务T1和T2读入同一数据并修改，T2的提交结果破坏了T1提交的结果，导致T1的修改被丢失。

修改-修改冲突

T1	T2
① $R(A)=16$	$R(A)=16$
②	
③ $A \leftarrow A-1$ $W(A)=15$	$A \leftarrow A-3$ $W(A)=13$
④	



2.不可重复读

■ 不可重复读是指事务T1读取数据后，事务T2 执行更新操作，使T1无法再现前一次读取结果。

■ 不可重复读包括三种情况：

(1) 情况1

- 事务1读取某一数据
- 事务2对其做了修改
- 当事务1再次读该数据时，得到与前一次不同的值

(2) 情况2

- 事务T1按一定条件从数据库中读取了某些数据记录
- 事务T2删除了其中部分记录
- 当T1再次按相同条件读取数据时，发现某些记录神秘地消失了。

(3) 情况3

- 事务T1按一定条件从数据库中读取某些数据记录
- 事务T2插入了一些记录
- 当T1再次按相同条件读取数据时，发现多了一些记录

T1	T2
① R(A)=50 R(B)=100 求和=150	R(A)=50
②	R(B)=100 B←B*2 W(B)=200
③ R(A)=50 R(B)=200 求和=250 (验算不对)	

后两种不可重复读有时也称为幻影现象（Phantom Row）

读-更新冲突

3.读“脏”数据

■ 读“脏”数据是指：

- 事务T1修改某一数据，并将其写回磁盘
- 事务T2读取同一数据后，T1由于某种原因被撤销
- 这时T1已修改过的数据恢复原值，T2读到的数据就与数据库中的数据不一致。
- T2读到的数据就为“脏”数据，即不正确的数据

修改-读冲突

T1	T2
① $R(C)=100$ $C \leftarrow C*2$ $W(C)=200$	$R(C)=200$
②	
③ ROLLBACK C恢复为100	



并发控制概述

- 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，可以降低对一致性的要求以减少系统开销
- 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)
- 商用的DBMS一般都采用封锁方法



11 并发控制

11.1 并发控制概述

11.2 封锁

11.3 封锁协议

11.4 活锁和死锁

11.5 并发调度的可串行性

11.6 两段锁协议

11.7 封锁的粒度

*11.8 其他并发控制机制

11.9 小结



11.2 封锁

■ 资源共享的方式， 直观解释

可被多人同时共享的资源：大屏幕电视，广播，Bus

可被分时共享的资源，(一个时刻只能被一个事务占用)：

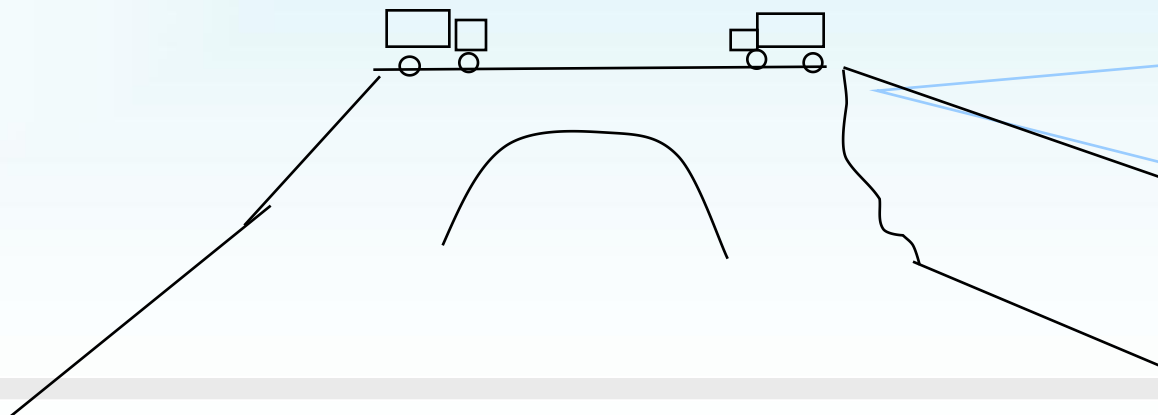
公用电话间，移动厕所，窄桥，选票

解决方法 LOCK(有人) / UNLOCK (无人)

解决冲突：按制度（协议） 分时 共享 资源.

说：不能七嘴八舌，听：可以七耳八朵

- 争夺资源
- 互相等待对方释放资源，引起死锁



封锁

■ 资源共享的方式， 直观解释

可被多人同时共享的资源：大屏幕电视，广播，Bus

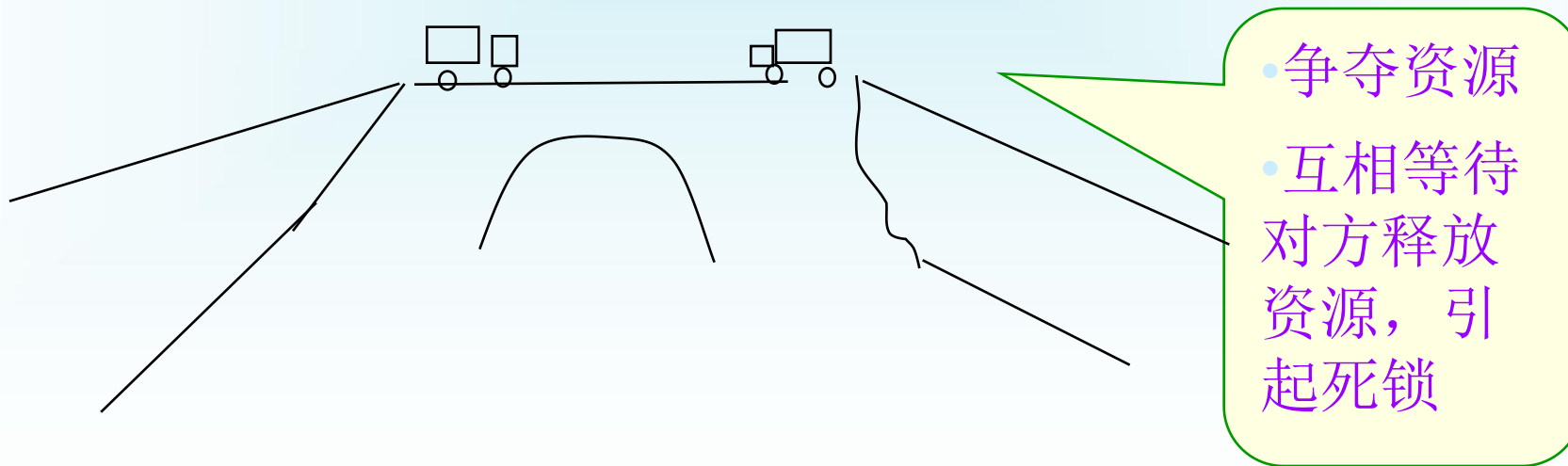
可被分时共享的资源，(一个时刻只能被一个事务占用)：

公用电话间，移动厕所，窄桥，选票

解决方法 LOCK(有人) / UNLOCK (无人)

解决冲突：按制度（协议） 分时 共享 资源.

说(写)：不能七嘴八舌，听（读）：可以七耳八朵



11.2 封锁



■ 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。

■ 基本封锁类型

排它锁（Exclusive Locks，简记为X锁）——写锁

- ✓ 其采用的原理是禁止并发操作。
- ✓ 事务T对数据对象A加了X锁，则允许T读取和修改A，但不允许其它事务再对A加任何类型的锁和进行任何操作。

T1	T2
① R(A)=16	R(A)=16
②	
③ $A \leftarrow A-1$	$A \leftarrow A-3$
 W(A)=15	
④	 W(A)=13



11.2 封锁

■ 基本封锁类型

排它锁 (Exclusive Locks, 简记为X锁) --写锁

共享锁 (Share Locks, 简记为S锁) --读锁

- ✓ 其采用的原理是允许其他用户对同一数据对象进行查询，但不能对该数据对象进行修改。
- ✓ 事务T对数据对象A加了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释放了A上的S锁。
- ✓ 保证了其他事务在T释放R上的S锁之前，只能读取R，而不能再对R作任何修改。



锁兼容性

■ 锁兼容性矩阵

T1 \ T2	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

- 如果事务对某数据项的锁请求与其他事务在该数据项上已有的锁兼容，则请求被批准
- 任意数目的事务可对同一数据持有共享锁；但若一事务对某数据持有排他锁，则其他事务不得持有该数据上的任何锁。
- 若不能授予锁，则使请求事务等待持有不兼容锁的所有其他事务释放锁。然后才可授予锁。



11 并发控制

- 11.1 并发控制概述
- 11.2 封锁
- 11.3 封锁协议
- 11.4 活锁和死锁
- 11.5 并发调度的可串行性
- 11.6 两段锁协议
- 11.7 封锁的粒度
- *11.8 其他并发控制机制
- 11.9 小结



11.3 封锁协议

■ 封锁协议

运用封锁方法时，对数据对象加锁时需要约定一些规则：

- 何时申请封锁？
- 持锁时间？
- 何时释放封锁等？

■ 三级封锁协议

并发操作所带来的丢失更新、读脏数据和不可重复读等数据不一致性问题，可以通过三级封锁协议在不同程度上给予解决。

- 一级封锁协议：事务T在写数据X前必须获得X的排它锁，事务结束时释放该锁。
- 二级封锁协议：一级封锁协议 + 事务T在读数据X前必须获得X的共享锁，读取后释放锁。
- 三级封锁协议：一级封锁协议 + 事务T在读数据X前必须获得X的共享锁，事务结束时释放该锁。



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。

■ 一级封锁协议只有当修改数据时才进行加锁，如果只是读取数据并不加锁，所以它不能保证不读“脏”数据和可重复读数据。



使用封锁机制解决丢失修改问题

T1	T2
① R(A)=16	
②	R(A)=16
③ A←A-1 W(A)=15	
④	A←A-3 W(A)=13

没有丢失修改

T1	T2
① Xlock A	
② R(A)=16	
	Xlock A
③ A←A-1 W(A)=15 Commit Unlock A	等待 等待 等待 等待
④	获得XlockA R(A)=15 A←A-3 W(A)=12 Unlock A



使用封锁机制解决丢失修改问题

T1	T2
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)	

T1	T2
① R(A)=50 R(B)=100 求和=150	
②	Xlock B 获得 R(B)=100 B←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

使用封锁机制解决丢失修改问题

T1	T2
① $R(C)=100$ $C \leftarrow C*2$ $W(C)=200$	
②	$R(C)=200$
③ ROLLBACK C恢复为100	

T1	T2
① Xlock C 获得	
② $R(C)=100$ $C \leftarrow C*2$ $W(C)=200$	
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

一级封锁协议加上事务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。

■ 在2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。



使用二级封锁协议解决丢失修改和读“脏”数据问题

- 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=14送回到磁盘。避免了丢失T1的更新。

没有丢失修改

T1	T2
① Xlock A	
② R(A)=16	
	Xlock A
③ $A \leftarrow A-1$ $W(A)=15$ Commit Unlock A	等待 等待 等待 等待
④	$R(A)=15$ $A \leftarrow A-3$
⑤	$W(A)=12$ Commit Unlock A

使用二级封锁协议解决丢失修改和读“脏”数据问题

T1	T2
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

T1	T2
① Xlock C R(C)=100 C←C*2 W(C)=200	
②	Slock C
③ ROLLBACK (C恢复为100) Unlock C	等待 等待 等待
④	获得Slock C R(C)=100
⑤	Commit C Unlock C

未读“脏”数据



使用二级封锁协议不能解决的问题：不可重复读

T1	T2
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)	

T1	T2
① Sclock A 获得 读A=50 Unlock A	
② Sclock B 获得	
③	Xlock B
④ 读B=100 Unlock B 求和=150	等待 等待
⑤	获得 读B=100 B←B*2 写回B=200 Commit

T1 (续)	T2
⑥ Sclock A 获得 读A=50 Unlock A Sclock B 获得 读B=200 Unlock B 求和=250 (验算不对)	

不可重复读

不可重复读



3. 三级封锁协议

■ 三级封锁协议

➤ 一级封锁协议加上事务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。



3. 三级封锁协议

T1	T2
① Xlock A	
② R(A)=16	
③	Xlock A
④ $A \leftarrow A-1$ W(A)=15 Commit Unlock A	等待 等待 等待 等待
⑤	获得Xlock A R(A)=15 $A \leftarrow A-3$
⑥	W(A)=12 Commit Unlock A

T ₁	T ₂
① Slock A 读A=50 Slock B 读B=100 求和=150	
②	Xlock B 等待 等待 等待 等待 等待 等待 等待 等待
③ 读A=50 读B=100 求和=150 Commit Unlock A Unlock B	
④	获得Xlock B 读B=100 $B \leftarrow B*2$ 写回B=200 Commit Unlock B
⑤	

可重复读

T ₁	T ₂
① Xlock C 读C= 100 $C \leftarrow C*2$ 写回C=200	
②	
③ ROLLBACK (C恢复为100) Unlock C	Slock C 等待 等待 等待 等待
④	获得Slock C 读C=100
⑤	Commit C Unlock C

不读“脏”数据



没有丢失修改

4. 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
- 何时释放锁（即持锁时间）

	X锁（对写数据）	S锁（对只读数据）	不丢失修改（写）	不读脏数据（读）	可重复读（读）
一级	事务全程加锁	不加	√		
二级	事务全程加锁	事务开始加锁，读完即放	√	√	
三级	事务全程加锁	事务全程加锁	√	√	√

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		√		√	√	√	√



11 并发控制

11.1 并发控制概述

11.2 封锁

11.3 封锁协议

11.4 活锁和死锁

11.5 并发调度的可串行性

11.6 两段锁协议

11.7 封锁的粒度

*11.8 其他并发控制机制

11.9 小结



11.4 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

- 活锁
- 死锁

T ₁	T ₂	T ₃	T ₄
Lock R	•	•	•
•	•	•	•
•	•	•	•
•	•	•	•
Unlock R	Lock R		
	等待	Lock R	
	等待	等待	
	等待	•	
	等待	Lock R	
•	等待	•	Lock R
•	等待	•	等待
•	等待	Unlock R	•
	等待	•	Lock R
	等待		•

活锁

T ₁	T ₂
Lock R ₁	•
	•
•	•
•	Lock R ₂
•	•
Lock R ₂	•
等待	
等待	
等待	
等待	Lock R ₁
等待	等待
•	•
•	•
•	•



如何避免活锁

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。



解决死锁的方法

■ 解决死锁的方法

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。

➤ 1. 死锁的预防

★ (1) 一次封锁法

★ (2) 顺序封锁法

➤ 2. 死锁的诊断与解除

T ₁	T ₂
<u>Lock R₁</u>	•
•	•
•	•
•	<u>Lock R₂</u>
Lock R ₂	•
等待	•
等待	
等待	
等待	Lock R ₁
等待	等待
•	等待
•	•
•	•



死锁的预防

- 预防死锁的发生就是要破坏产生死锁的条件
- 预防死锁的方法
 - 一次封锁法
 - 顺序封锁法



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
- 一次封锁法存在的问题：降低并发度
 - 扩大封锁范围
 - 将以后要用到的全部数据加锁，势必扩大了封锁的范围，从而降低了系统的并发度
- 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
 - 解决方法：将事务在执行过程中可能要封锁的数据对象全部加锁，这进一步降低了并发度。



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
 - 维护成本高
 - 数据库系统中可封锁的数据对象极其众多，并且随数据的插入、删除等操作而不断地变化，要维护这样极多而且变化的资源的封锁顺序非常困难，成本很高，难于实现
 - 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。

例：规定数据对象的封锁顺序为A,B,C,D,E。事务T3起初要求封锁数据对象B,C,E，但当它封锁了B,C后，才发现还需要封锁A，这样就破坏了封锁顺序。



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很适合数据库的特点
- **DBMS**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



死锁的诊断与解除

- 允许死锁发生
- 解除死锁
 - 由DBMS的并发控制子系统定期检测系统中是否存在死锁
 - 一旦检测到死锁，就要设法解除



检测死锁：超时法

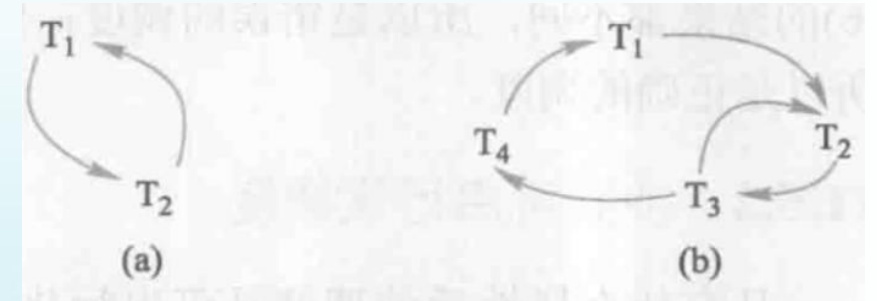
- 如果一个事务的等待时间超过了规定的时限，就认为发生了死锁
- 优点：实现简单
- 缺点
 - 有可能误判死锁
 - 时限若设置得太长，死锁发生后不能及时发现



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



■ 并发控制子系统周期性地（比如每隔1 min）检测事务等待图，如果发现图中存在回路，则表示系统中出现了死锁。

■ 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。



11 并发控制

11.1 并发控制概述

11.2 封锁

11.3 封锁协议

11.4 活锁和死锁

11.5 并发调度的可串行性

11.6 两段锁协议

11.7 封锁的粒度

*11.8 其他并发控制机制

11.9 小结



并发调度的正确性准则

■ 如何知道或判定并发执行后结果是否正确呢？

■ 基本概念

- 并发调度原则：既要交错执行，以充分利用系统资源；又要避免访问冲突。
- 事务调度：是一串事务中所有操作的顺序序列。
- 事务调度原则：调度中，不同事务的操作可以交叉，但需保持各个事务的操作次序。



11.5 并发调度的可串行性

- 将所有事务串行起来的调度策略一定是正确的调度策略。
- 所有的串行调度都被认为是正确的，尽管串行调度可能产生不同的结果，但是它从不会使数据库处于不一致的状态。
- 几个事务的并行执行是正确的，当且仅当其结果与按某一次序串行地执行它们时的结果相同。这种并行调度策略称为可串行化（**Serializable**）的调度。
- 正确调度：可串行化的调度。
- 等价调度:如有两个调度 S_1 和 S_2 ，在DB的任一初始状态下，所有读出的数据都是一样的，留给DB的最终状态也是一样的，则称 S_1 和 S_2 是等价的。



什么样的并发操作调度是正确的

■ 可串行性是并行事务正确性的唯一准则

例：现在有两个事务，分别包含下列操作：

事务1：读B； $A=B+1$ ；写回A；

事务2：读A； $B=A+1$ ；写回B；

假设A的初值为2，B的初值为2。

➤ 对这两个事务的不同调度策略

★ 串行执行

- 串行调度策略a
- 串行调度策略b

★ 交错执行

- 不可串行化的调度c
- 可串行化的调度d



串行调度策略，正确的调度

(a)

T ₁	T ₂
Slock B Y=B=2 Unlock B Xlock A A=Y+1 写回A(=3) Unlock A	Slock A X=A=3 Unlock A Xlock B B=X+1 写回B(=4) Unlock B

(b)

T ₁	T ₂
Slock B Y=B=3 Unlock B Xlock A A=Y+1 写回A(=4) Unlock A	SlockA X=A=2 Unlock A Xlock B B=X+1 写回B(=3) Unlock B



不可串行化的调度

(c)	T_1	T_2
	Slock B $Y=B=2$	
		Slock A $X=A=2$
	Unlock B	
	Xlock A $A=Y+1$ 写回A(=3)	Unlock A
		Xlock B $B=X+1$ 写回B(=3)
	Unlock A	Unlock B

由于其执行结果与(a)、(b)的结果都不同，所以是错误的调度。



可串行化的调度

(d)

T_1	T_2
Slock B	
$Y=B=2$	
Unlock B	
Xlock A	
$A=Y+1$	Slock A
写回A(=3)	等待
Unlock A	等待
	等待
	$X=A=3$
	Unlock A
	Xlock B
	$B=X+1$
	写回B(=4)
	Unlock B

由于其执行结果与串行调度（a）的执行结果相同，所以是正确的调度。



冲突可串行化调度

■ 冲突操作

- 冲突操作是指不同的事务对同一个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 */
 - $W_i(x)$ 与 $W_j(x)$ /* 事务 T_i 写 x , T_j 写 x */
- 其他操作是不冲突操作
- 不同事务的冲突操作和同一事务的两个冲突操作不能交换(Swap)



冲突可串行化调度（续）

■ 可串行化调度的充分条件

一个调度S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'，如果S'是串行的，称调度S为冲突可串行化的调度

■ 一个调度是冲突可串行化，一定是可串行化的调度



冲突可串行化调度（续）

- 一个调度是冲突可串行化，一定是可串行化的调度

[例] 今有调度 $Sc1=r1(A)w1(A)r2(A)\underline{w2(A)}\underline{r1(B)w1(B)}r2(B)w2(B)$

- 把 $w2(A)$ 与 $r1(B)w1(B)$ 交换，得到：

$r1(A)w1(A)\underline{r2(A)}\underline{r1(B)w1(B)}\underline{w2(A)}r2(B)w2(B)$

- 再把 $r2(A)$ 与 $r1(B)w1(B)$ 交换：

$Sc2=r1(A)w1(A)r1(B)w1(B)\underline{r2(A)w2(A)}r2(B)w2(B)$

- $Sc2$ 等价于一个串行调度 $T1, T2$, $Sc1$ 冲突可串行化的调度



冲突可串行化调度（续）

- 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]有3个事务

$T1=W1(Y)W1(X)$ ， $T2=W2(Y)W2(X)$ ， $T3=W3(X)$

- 调度 $L1=W1(Y)\underline{W1(X)}W2(Y)W2(X)\underline{W3(X)}$ 是一个串行调度。
- 调度 $L2=W1(Y)W2(Y)W2(X)\underline{W1(X)}W3(X)$ 不满足冲突可串行化。但是调度 $L2$ 是可串行化的，因为 $L2$ 执行的结果与调度 $L1$ 相同， Y 的值都等于 $T2$ 的值， X 的值都等于 $T3$ 的值



如何保证并发操作的调度是正确的

- 并发操作三种不一致原因：是由于调度是不可串行化而导致的。并发操作破坏了事务的隔离性。要采用正确的调度方式，使一个事务不受其它事务的干扰。达到可串行化的结果
- 能否找到一种方法或策略控制事务调度，使其满足可串行化？
- 如何通过加锁技术保证事务调度的可串行化？
- 保证 冲突可串行调度
- 保证并发操作调度正确性的方法
 - 基于锁(locking)的协议:两段锁（Two-Phase Locking, 简称2PL）协议
 - 基于时间戳(timestamping)的协议



11.6 两段锁协议

- 数据库管理系统普遍采用两段锁协议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（Two-Phase Locking，简称2PL）的含义是：
 1. 申请封锁期（开始对数据操作之前）：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
 2. 释放封锁期（结束对数据操作之后）：在释放一个封锁之后，事务不再获得任何其他封锁
 - 两段锁协议是保证并发调度的可串行性的封锁协议
 - 两段锁协议是实现可串行化调度的充分条件。



两段锁协议（续）

■ “两段”锁的含义

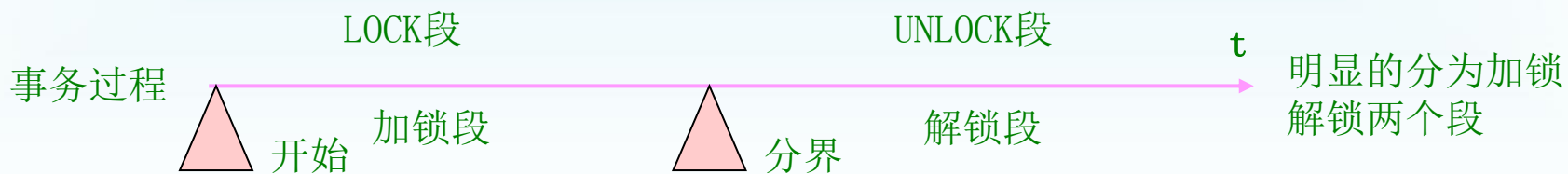
➤ 阶段1: 扩展（增长）阶段

- ★ 事务 可获得锁
- ★ 事务不能释放锁

➤ 阶段2: 收缩阶段

- ★ 事务可释放锁
- ★ 事务不能获得锁

- 本协议确保可串行化. 可以证明事务可按它们的**lock points** (即事务获得最后一个锁的点)的次序串行化.



两段锁协议（续）

如果事务中所有的加锁操作都在事务的第一个解锁操作之前进行，那么这个事务是遵循两段锁协议的。

具体如下：

- 事务在对一个数据项进行操作之前，必须先获得对该数据项的锁。根据访问类型，锁可以是读或写锁。
- 一旦事务释放了一个锁，它就不能再获得任何新锁。

扩展（**增长阶段**），事务获得它所需要的所有锁（不一定是同时），但不释放其中任何一个；**收缩阶段**，在这个阶段，事务释放它所拥有的锁，但不能在请求任何新锁。



两段锁协议（续）

例：

事务1的封锁序列：

Slock A ... Slock B ... Xlock C ... Unlock B ... Unlock A ... Unlock C;

事务2的封锁序列：

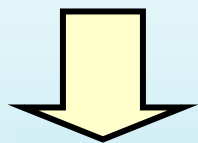
Slock A ... Unlock A ... Slock B ... Xlock C ... Unlock C ... Unlock B;

事务1遵守两段锁协议，而事务2不遵守两段协议。



两段锁协议（续）

- 并行执行的所有事务均遵守两段锁协议，则对这些事务的所有并行调度策略都是可串行化的。



所有遵守两段锁协议的事务，其并行执行的结果一定是正确的

- 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
- 可串行化的调度中，不一定所有事务都必须符合两段锁协议。



两段锁协议（续）

T ₁	T ₂	T ₁	T ₂	T ₁	T ₂
Slock B		Slock B			Slock A
读B=2		读B=2			读A=2
Y=B		Y=B			X=A
Xlock A		Unlock B			Unlock A
	Slock A	Xlock A		Slock B	
	等待		Slock A	读B=2	
	等待		等待	Y=B	Xlock B
A=Y+1	等待	A=Y+1	等待	Unlock B	等待
写回A=3	等待	写回A=3	等待		Xlock B
Unlock B	等待	Unlock A			B=X+1
Unlock A	Slock A		Slock A		写回B=3
	读A=3		读A=3		Unlock B
	Y=A		X=A		
	Xlock B		Unlock A	Xlock A	
	B=Y+1		Xlock B	A=Y+1	
	写回B=4		B=X+1	写回A=3	
	Unlock B		写回B=4	Unlock A	
	Unlock A		Unlock B		

(a) 遵守两段锁协议

(b) 不遵守两段锁协议

(c) 不遵守两段锁协议



两段锁协议（续）

■ 两段锁协议与防止死锁的一次封锁法

- 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

T_1	T_2
Slock B 读B=2	
	Slock A 读A=2
Xlock A 等待 等待	Xlock B 等待



两段锁协议（续）

T1	T2
① Xlock A	
② R(A)=16	
③	Xlock A
④ A←A-1 W(A)=15 Commit Unlock A	等待 等待 等待 等待
⑤	获得Xlock A R(A)=15 A←A-3
⑥	W(A)=12 Commit Unlock A

T ₁	T ₂
① Slock A 读A=50 Slock B 读B=100 求和=150	
②	Xlock B 等待 等待 等待 等待 等待 等待 等待
③ 读A=50 读B=100 求和=150 Commit Unlock A Unlock B	
④	获得Xlock B 读B=100 B←B*2 写回B=200 Commit Unlock B
⑤	
可重复读	

T ₁	T ₂
① Xlock C 读C= 100 C←C*2 写回C=200	
②	
③ ROLLBACK (C恢复为100) Unlock C	Slock C 等待 等待 等待 等待
④	获得Slock C 读C=100
⑤	Commit C Unlock C

没有丢失修改

不读“脏”数据



11 并发控制

- 11.1 并发控制概述
- 11.2 封锁
- 11.3 封锁协议
- 11.4 活锁和死锁
- 11.5 并发调度的可串行性
- 11.6 两段锁协议
- 11.7 封锁的粒度
- *11.8 其他并发控制机制
- 11.9 小结



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：在关系数据库中，封锁对象：

- 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T1需要修改元组L1，则T1必须对包含L1的整个数据页A加锁。如果T1对A加锁后事务T2要修改A中元组L2，则T2被迫等待，直到T1释放A。
- 如果封锁粒度是元组，则T1和T2可以同时为L1和L2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

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



选择封锁粒度的原则（续）

■ 选择封锁粒度的原则

- 封锁的粒度越 大，小，
- 系统被封锁的对象 少，多，
- 并发度 小，高，
- 系统开销 小，大，
- 选择封锁粒度：
 - ✓ 考虑封锁机制和并发度两个因素
 - ✓ 对系统开销与并发度进行权衡

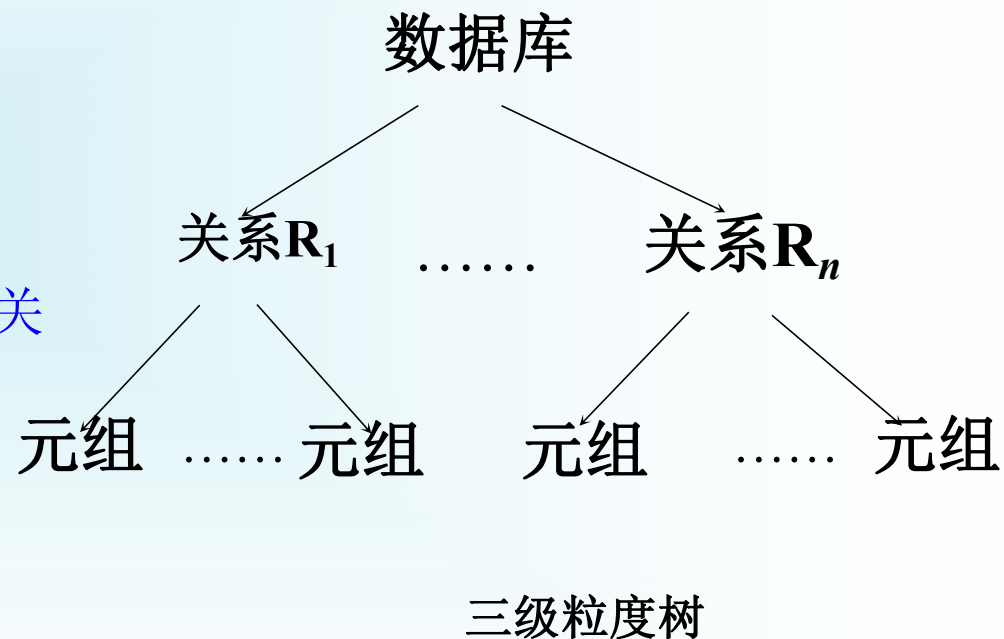


多粒度封锁

■ 多粒度树

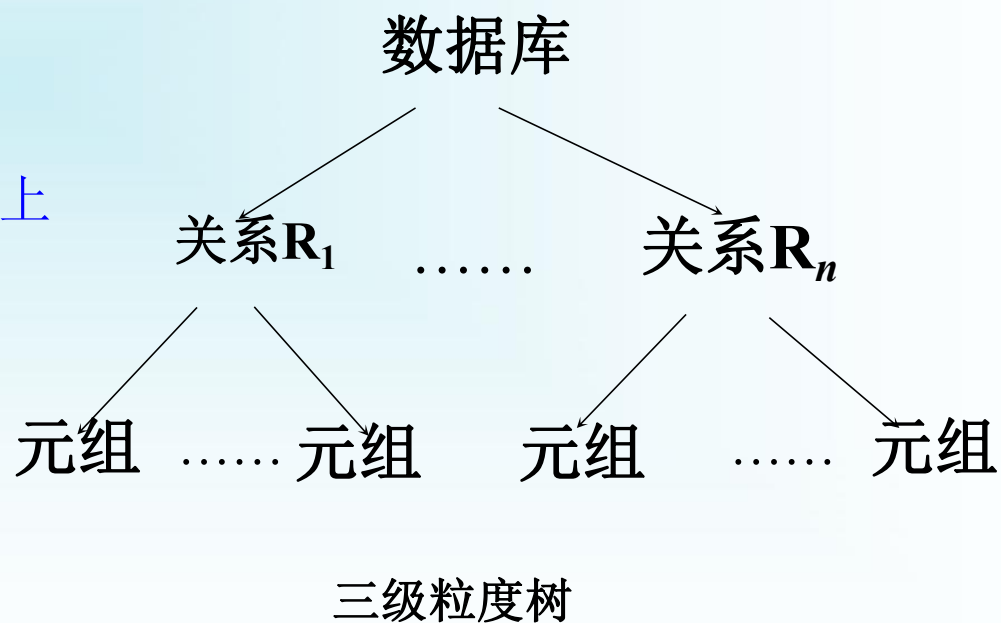
- 以树形结构来表示多级封锁粒度
- 根节点是整个数据库，表示最大的数据粒度
- 叶结点表示最小的数据粒度

- 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。



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: 直接加到数据对象上的封锁
- 隐式封锁: 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

- 系统检查封锁冲突时
 - 要检查显式封锁
 - 还要检查隐式封锁
- 例如事务T要对关系R1加X锁
 - 系统必须搜索其上级结点数据库、关系R1
 - 还要搜索R1的下级结点，即R1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



常用意向锁

■ 意向共享锁(Intent Share Lock, 简称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T1要对R1中某个元组加S锁, 则要首先对关系R1和数据库加IS锁

■ 意向排它锁(Intent Exclusive Lock, 简称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T1要对R1中某个元组加X锁, 则要首先对关系R1和数据库加IX锁

■ 共享意向排它锁(Share Intent Exclusive Lock, 简称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

$T_1 \backslash T_2$	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

Y=Yes，表示相容的请求

N=No，表示不相容的请求

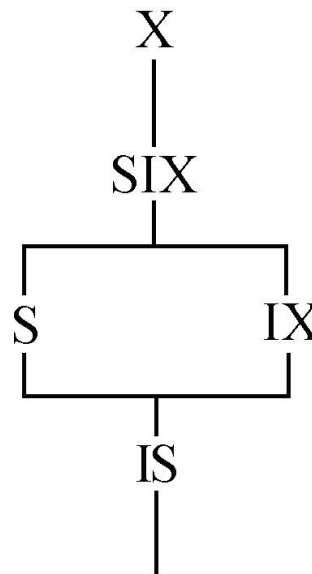
(a) 数据锁的相容矩阵



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T1要对关系R1加S锁

- 要首先对数据库加IS锁
- 检查数据库和R1是否已加了不相容的锁(X或IX)
- 不再需要搜索和检查R1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



11.9 小结

■ 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)

■ 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)
- 多粒度封锁 (意向锁)
- 活锁和死锁

■ 解决数据不一致的并发控制协议：三级封锁协议

■ 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



作业

■ P326: 1、2、4、8



下课了。。



休息。。。

