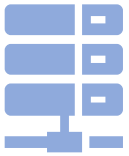
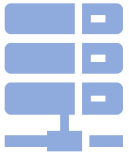


第九章 并发控制



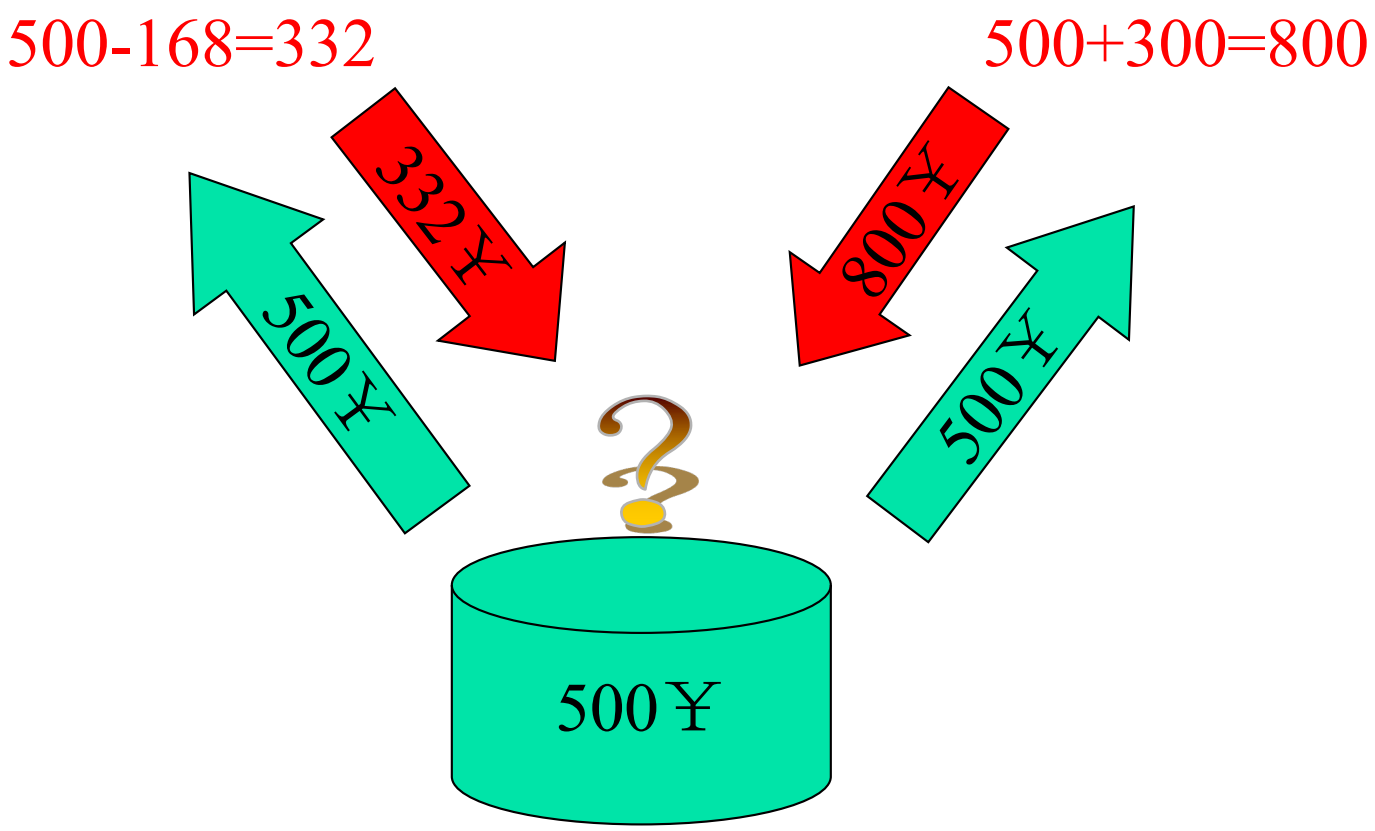
并发控制

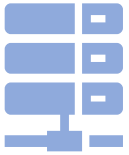
■ 数据库的并发性

- 数据库的最大特点之一就是数据资源是共享的
- 每个用户在存取数据库中的数据时，可能是串行执行，即每个时刻只有一个用户程序运行，也可能是多个用户并行地存取数据库。
- 串行执行意味着一个用户在运行程序时，其他用户程序必须等到这个用户程序结束才能对数据库进行存取，这样如果一个用户程序涉及大量数据的输入/输出交换，则数据库系统的大部分时间将处于闲置（只对一个用户提供服务）状态。
- 因此，为了充分利用数据库资源，数据库用户都是对数据库系统并行存取数据

并发控制

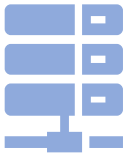
■ 当多个用户同时存取、修改DB中的数据时，可能会发生相互干扰，使DB中数据的完整性受到破坏，而导致数据不一致，数据库不可用





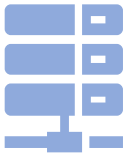
并发控制

- 数据库的一致性（Consistency）
  - 当多个用户并发存取同一数据块时，如果对并发操作不加控制可能会产生不正确的数据，破坏数据的一致性
  - 数据的一致性：在任何时刻用户面对的数据库都是符合现实世界的语义逻辑的。
    - 除并发操作外，数据库软硬件故障也可能破坏一致性
  - DBMS必须对并发操作进行控制，避免破坏数据一致性
  - 数据库的并发控制是以**事务**为基本单位进行的，通过对事务所操作的数据施加封锁来实现一致性
    - 这种数据一致性约束属于动态关系约束



事务

- 事务(Transaction)是用户定义的一个数据库操作序列，这些操作要么全做，要么全不做，是一个不可分割的工作单位
  - 银行转账， 批量成绩修改，车票销售 …
- 事务和程序是两个概念
  - 一个事务可以是一条SQL语句或一组连续的SQL语句
    - 如果都是单条语句会产生一致性冲突吗?
  - 一个对数据库进行操作的应用程序可以包含多个事务，而这些事务是串行的



事务

■ SQL中事务的定义

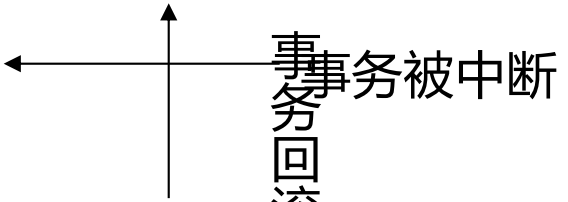
- 事务的开始与结束可以由用户显式控制。如果用户没有显式地定义事务，则由DBMS按照缺省规定自动划分事务（通常一条DML语句为一个事务）
- 用户自定义的事务以SQL语句Begin transaction开始，以Commit 或 Rollback 结束。
  - ✓ COMMIT表示事务的提交，即将事务中所有对数据库的更新写回到磁盘上的物理数据库中去，此时事务正常结束；
  - ✓ ROLLBACK表示事务的回滚，即在事务运行的过程中发生了某种故障，事务不能继续执行，系统将事务中对数据库的所有已完成的更新操作全部撤销，再回滚到事务开始时的状态

事务

■一个事务的例子

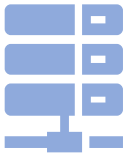
银行转帐：事务T从A帐户过户100￥到B帐户。

- 事务开始
- 系统从甲在A银行帐户上删除100元
- 系统在B银行帐户上增加100元
- 事务结束



事务代码：

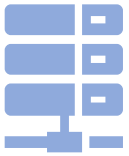
```
BEGIN TRANSACTION
  READ A
  A=A - 100
  IF A<0              /* A 款不足*/
 THEN
     BEGIN
          DISPLAY “A款不足”
               ROLLBACK
     END
ELSE     /* 拨款 */
    BEGIN
          READ B
          B=B+100
          DISPLAY “拨款完成”
          COMMIT
    END
```



事务

■一个事务的例子

- 该事务有两个出口：
  - ✓ 当A 帐号的款项不足时，事务以ROLLBACK（撤销）命令结束，即撤销该事务的影响；
  - ✓ 另一个出口是以COMMIT（提交）命令结束，完成从帐号A到帐号B的拨款。
  - ✓ 在COMMIT之前，即在数据库修改过程中，数据可能是不一致的，事务本身也可能被撤销。
  - ✓ 只有在COMMIT之后，事务操作引起的数据库变化才对其他事务开放，这就可以避免其他事务访问不一致或不存在的数据。



事务

■引入“事务”做为数据库逻辑操作单位的作用

- 做为并发控制的基本单位
- 遇到各类数据库错误（包括软硬件故障）后进行数据恢复的处理单位

■事务的管理需求

■事务是由有限的数据库操作序列组成，但并不是任意的数据库操作序列都能成为事务

■为了保护数据的完整性，一般要求事务具有**ACID**四个特征

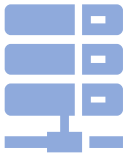
事务

■原子性（**A**tomicity）

- 管理需求
  - ✓ 如果事务因为软硬件故障在第三步和第六步之间终止退出了，部分操作完成而部分操作未完成，则数据库会处于不一致状态，钱会丢失
- 原子性特性
  - ✓ 事务中包含的所有操作要么全做，要么全不做。
  - ✓ 系统应保证只执行了一部分的事务不会对数据库状态产生影响

转账的事务T1：

1. **read**(*A*)
2. *A* := *A* – *100*
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + *100*
6. **write**(*B*)

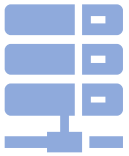


事务

■隔离性（Isolation）

- 管理需求
  - ✓ 如果在第三步和第六步中间，另一个事务T2被允许访问已被事务T1部分修改了的数据库，则它看到的是一个“不一致”的数据库
- 隔离性特性
  - ✓ 并发执行的各个事务之间不能互相干扰
  - ✓ 一个事务内部的操作及使用的数据对其他并发事务是隔离的
  - ✓ 每个事务在执行的时候应感觉不到其它事务的存在

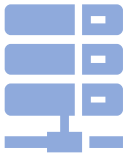
	T1	T2
1.	**read**(*A*)	
2.	*A* := *A* – *100*	
3.	**write**(*A*)	
	<--------	**read(B)**
4.	**read**(*B*)	
5.	*B* := *B* + *100*	
6.	**write**(*B*)	
7.		**read(B)**



事务

■一致性（Consistency）

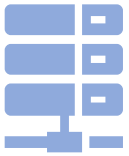
- 管理需求
  - ✓ 隐含的完整性约束条件，需要由用户根据业务规则在事务代码中做出“一致性”的规定（用户需自己保证一致性的定义和业务检查逻辑是正确的）
  - ✓ 如在转账例子中，A和B账户的金额总和在事务执行前后不能发生改变
- 一致性特性
  - ✓ 事务的隔离执行必须保证数据库的一致性。假定事务的一致性规定是正确的，则数据库在事务执行前是一致的，执行过程中可能暂时出现不一致，而当事务执行完后，数据库仍会处于一致性状态



事务

■持久性（Durability）

- 管理需求
  - ✓ 一旦用户得到事务已经执行完成的通知（如100元转账已经完成），则意味着事务对数据库的更新已被持久的记录下来，即使之后发生了系统软、硬件失效，也不能抹除事务的执行结果
- 持久性特性
  - 一个事务一旦提交（commit）之后，它对数据库的影响必须是永久的。系统发生失效不能改变事务的持久性。

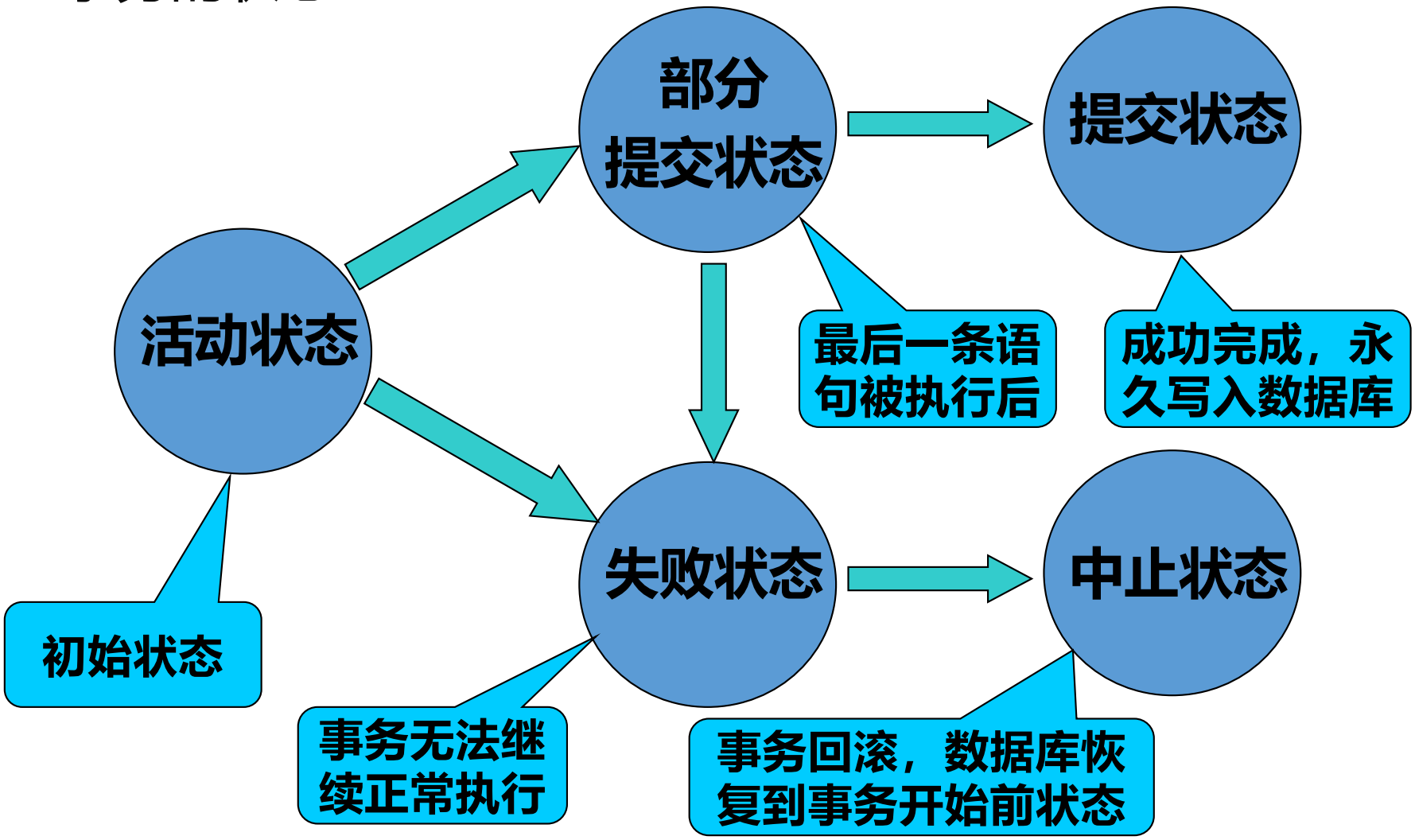


事务

- 事务的执行状态
  - 活动状态（Active） – 是事务的初始状态；事务在执行的时候处于这种状态
  - 部分提交状态（Partially committed） – 对数据库进行了部分更新，但事务并未最后结束的状态
  - 失败状态（Failed) – 由于物理或逻辑错误导致事务语句无法进行下去的状态
  - 中止状态（Aborted） – 事务回滚所有操作，数据库恢复到事务开始之前的状态
  - 提交状态（Committed） – 事务正确执行完成的状态.

事务

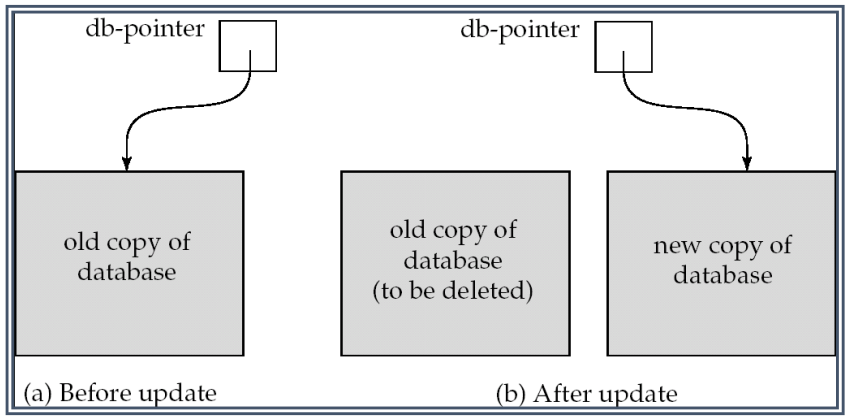
■事务的状态

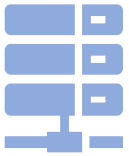

部分
提交状态
提交状态
活动状态
最后一条语
句被执行后
成功完成，永
久写入数据库
初始状态
失败状态
中止状态
事务无法继
续正常执行
事务回滚，数据库恢
复到事务开始前状态

事务

■数据库怎么保证事务特性的实现?

- 数据库的恢复机制保证了原子性和持久性的实现
- 一种简单的恢复机制： shadow-database影子数据库技术
  - ✓ 假定同时只有一个事务是活动的（不存在隔离性问题）
  - ✓ 对数据库的更新都在影子数据库上进行。当事务更新都完成后，数据库指针db-pointer指向影子数据库，将之变成当前数据库。若事务出现错误，则删除影子数据库
- 恢复机制不能解决事务并发执行情况下的数据一致性、隔离性问题

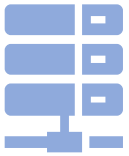




事务

■事务的并发执行

- 数据库中时常运行多个事务。事务的执行可以是并行的，也可以是串行的
- 并行 Vs 串行
  - ✓ 串行事务效率低，而并行事务会破坏数据库的一致性
- 并行的优点
  - ✓ 一个事务由不同的步骤组成，所涉及的系统资源也不同。这些步骤可以并发执行，以提高系统的吞吐量。
  - ✓ 系统中存在着周期不等的各种事务，串行会导致难于预测的时延。如果各个事务所涉及的是数据库的不同部分，采用并发会减少平均响应时间。



事务

■事务的并发执行

- 当同一数据库系统中有多个事务并发运行时，如果不加以适当控制，可能会破坏事务的隔离性，产生数据的不一致性。
- 并发操作带来的数据不一致性
  - ✓ 丢失修改（lost update）
  - ✓ 不可重复读（non-repeatable read）
  - ✓ 读“脏”数据（dirty read）

事务

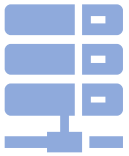
■事务的并发执行

- 丢失修改（lost update）
  - ✓ 丢失修改是指事务1与事务2从数据库中读入同一数据并修改，事务2的提交结果破坏了事务1提交的结果，导致事务1的修改被丢失。
  - ✓ 也称脏写（dirty write）

写写冲突

T_1	T_2
① 读A=16	
②	读A=16
③ A←A-1	
④写A=15	
	A←A-1
	写A=15

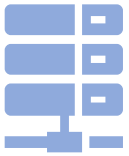
丢失修改



事务

■事务的并发执行

- 不可重复读（non-repeatable read）
  - ✓ 不可重复读是指事务1读取数据后，事务2执行更新操作，使事务1无法再现前一次读取结果。
- 三种不可重复读：事务1读取某一数据后
  - ✓ 1. 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. 事务2删除了其中部分记录，当事务1再次读取数据时，发现某些记录神密地消失了。
  - ✓ 3. 事务2插入了一些记录，当事务1再次按相同条件读取数据时，发现多了一些记录。
  - ✓ 后两种不可重复读有时也称为幻影现象（phantom row）



事务

■事务的并发执行

读写冲突

- 不可重复读（non-repeatable read）

T_1	T_2
① 读A=50 读B=100 求和=150	
②	读B=100 B←B*2 写B=200
③ 读A=50 读B=200 求和=250 (验算不对)	

T_1	T_2
① 读A=50 读B=100 求和=150	
②	删除B
③ 读A=50 未读取到B	

事务

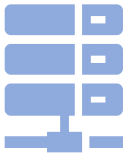
■事务的并发执行

- 读“脏”数据（dirty read）
  - ✓ 事务1修改某一数据，并将其写回磁盘。事务2读取同一数据后，事务1由于某种原因被撤消，这时事务1已修改过的数据恢复原值，事务2读到的数据就与数据库中的数据不一致，是不正确的数据，又称为“脏”数据。

写读冲突

T_1	T_2
① 读C=100 C←C*2 写C	
②	读C=200
③ ROLLBACK C恢复为100	C=200为脏数据

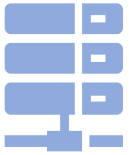
读脏数据



事务

■事务的并发控制机制

- 对事务的并行运行顺序进行合理安排，达成事务的隔离性和一致性
- 主要手段：
  - ✓ 调度
  - ✓ 锁
  - ✓ 协议



调度

■调度（Schedule）

- 调度是并发运行的事务中各条指令的执行序列
  - ✓ 调度应对各事务所有指令的执行时间顺序做出安排
  - ✓ 调度应保留同一事务中各指令原有的执行顺序
- 对并行事务中并行操作的调度有很多方案，而不同的调度方案可能会产生不同的结果。
- 将所有事务串行起来的调度策略一定是正确的调度策略。
  - ✓ 如果一个事务运行过程中没有其他事务在同时运行，也就是说它没有受到其他事务的干扰，那么就可以认为该事务的运行结果是正常的或者是预想的
  - ✓ 以不同的顺序串行执行事务也有可能会产生不同的结果，但由于不会将数据库置于不一致状态，所以都可以认为是正确的。

调度

- 业务描述：
  - T1事务：从A账户向B账户转账50元
  - T2事务：从A账户向B账户转账存款余额的10%
- 调度1：

一种串行化调度：

T1 → T2

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

调度

- 调度2:

另一种串行化调度:

T2 → T1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

调度

- 调度3：

  并不是串行调度

  但是执行效果等价于调度1

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

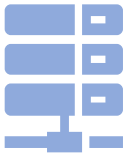
在调度1、2、3中，各事务执行前后的账户总金额A+B都是一致的

调度

- 调度4:

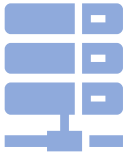
  该调度并没有保持账户总金额A+B的一致性

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)



调度

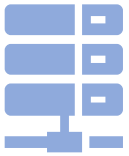
- 并行调度的可串行化
  - 串行调度能够保证数据库的一致性
    - ✓ 假定每个独立事务都能保证一致性，则这些事务的串行执行也就能保证一致性
  - 几个事务的并行执行是正确的，当且仅当其结果与按某一次序串行地执行它们时的结果相同。这种并行调度策略称为可串行化（Serializable）的调度



调度

- 并行调度的可串行化
  - 可串行化是并行事务正确性的唯一准则
  - 那怎么判断某个并行调度是可串行化的?
    - ✓ 可分为冲突可串行化（**conflict serializability**）和视图可串行化（**view serializability**）等
    - ✓ 我们只讨论冲突可串行化
  - 延伸阅读

    https://zhuanlan.zhihu.com/p/539168364



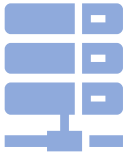
调度

■ 冲突可串行化

- 现有事务 T_i 和 T_j 及各自的指令 I_i 和 I_j，当且仅当有一个数据项 Q 被 I_i 和 I_j 都访问，且至少有一个指令对 Q 进行写入操作的时候，两个指令发生冲突。

1. I_i = **read**(Q), I_j = **read**(Q). I_i 和 I_j 不冲突
2. I_i = **read**(Q), I_j = **write**(Q). 冲突
3. I_i = **write**(Q), I_j = **read**(Q). 冲突
4. I_i = **write**(Q), I_j = **write**(Q). 冲突

- 直观上，I_i 和 I_j 的冲突状态规定了这两个指令执行上的时间顺序
  - ✓ 如果两个指令不冲突，则随便调换他们的执行顺序不会影响最终执行结果



调度

- 冲突可串行化
  - 如果一个调度S可以通过一系列的非冲突指令顺序调换而被转换为另一个调度S’， 则我们说S和S’是冲突等价的（conflict equivalent）
  - 如果S’是一个串行化调度，这时我们说S是冲突可串行化的（conflict serializable）

调度

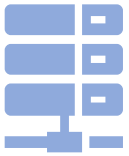
- 冲突可串行化
  - 调度5 可以转换为串行调度调度6，它是冲突可串行化的

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 5

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6



调度

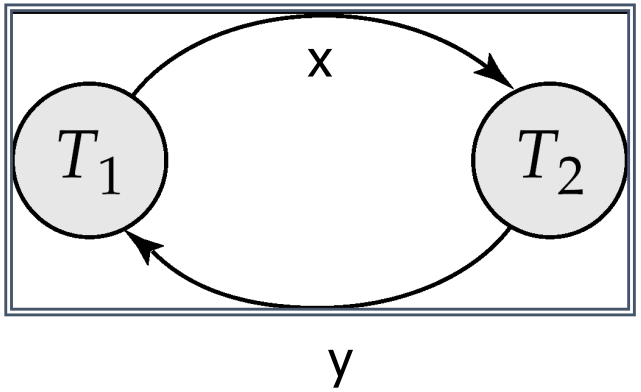
- 冲突不可串行化的例子
  - 我们对下面的调度7无法通过指令调换来获得等价的串行调度 < T_3, T_4 > , 或< T_4, T_3 > 。

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Schedule 7

调度

- 前趋图（Precedence graph）
  - 是一种有效的冲突可串行化的测试方法
  - 是一种有向图
    - ✓ 顶点是事务的名称
    - ✓ 对于两个顶点：事务 T_i 和 T_j，如果这两个事务冲突，且是事务 T_i 先访问的冲突资源，则画一个从 T_i 到 T_j 的弧。
    - ✓ 可以在弧上标注冲突资源的名称

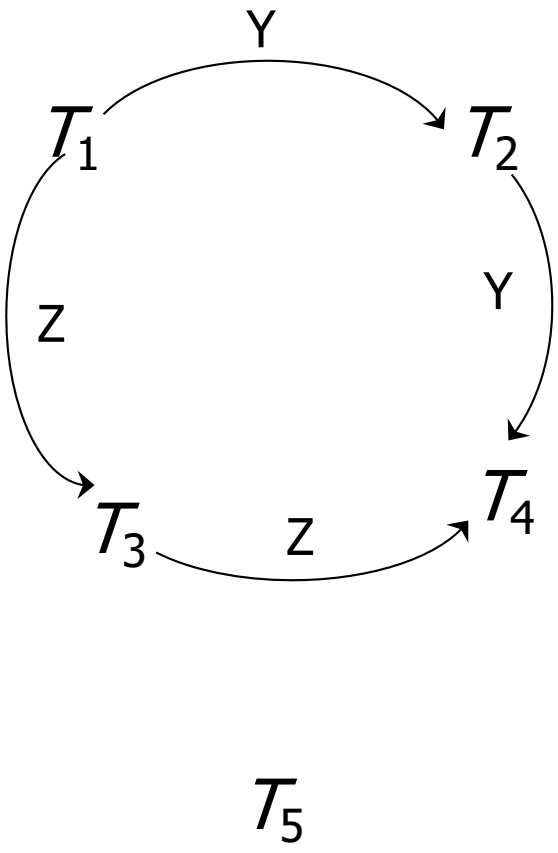


调度

前趋图示例

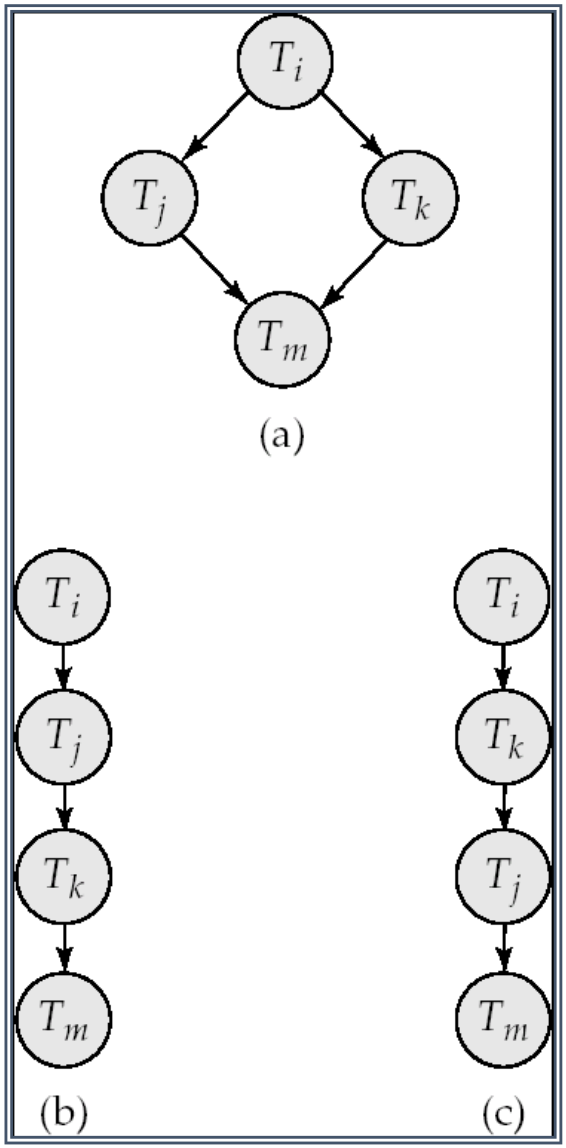
Schedule 8

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



调度

- 用前趋图检查冲突可串行性
  - 当且仅当前驱图是无环（acyclic）的时候，对应的调度是冲突可串行的
  - 当前驱图是无环的时候，冲突等价的串行化调度的事务执行顺序可以通过对图的拓扑排序（*topological sorting*）方法获得
  - 例如通过拓扑排序，可知前例调度8的串行化调度顺序可以是：$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$



调度

- 进一步考察若事务失败，对调度效果产生的影响
  - 对于调度9中的事务T9，若它在read（A）之后马上提交（Commit），则若T8在read（B）之后回滚（rollback），会导致数据库处于不一致状态
- 可恢复的调度（Recoverable Schedules）
  - 若事务T_j读取了一个事务T_i之前写入的数据，
  - 则T_i的提交操作应在T_j的提交操作之前发生
  - 可以避免前述数据回滚导致的不一致问题

需考虑当数据库发生故障后，事务的回滚对调度产生的影响

调度9

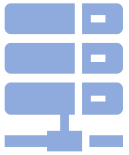
T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

调度

- 另一种情况：级联回滚（Cascading rollback）
  - 调度10是可恢复的调度。但若T10失败了，则它需要回滚。而T10的回滚会引起T11、T12的一连串回滚。这会导致事务运行不畅，数据库浪费大量计算资源
- 无级联调度（Cascadeless Schedules）
  - 每一对事务若事务T_j读取了一个事务T_i之前写入的数据，则T_i的提交操作应在T_j的读取操作之前发生
  - 无级联调度也是可恢复调度
  - 达到无级联调度是一种较高的调度要求

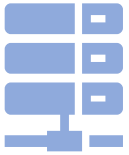
调度10

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)



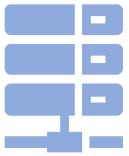
并发控制

- 数据库需要提供一种机制，来保证对并发事务实施的调度是
  - 冲突可串行的，可恢复的，最好是无级联的
- 一种策略是要求同一时间只有一个事务能运行，这样会强制产生串行调度，但是运行效率太低
- 在随机的产生一种调度之后，再用前驱图测试一下它是否是冲突可串行的？实际应用中不可行
- 因此需要设计一种控制策略，在接收到每一条新的事务指令的时候，能够根据当前事务运行和资源占用情况立即决定其执行的时机，且由此产生的调度是好的调度



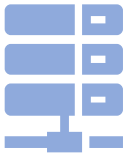
并发控制

- 保证并发调度正确性的方法：
  - 封锁方法（Locking）
    - ✓ 事务锁定使用的资源以防止其它事务访问 /修改
  - 时标方法（Time-Stamping ）
    - ✓ 为每一个事务分配一个全局唯一的时间戳
  - 乐观方法
    - ✓ 假定大多数数据库操作是不冲突的
  - 多版本并发控制
    - ✓ 维护数据对象的多个版本



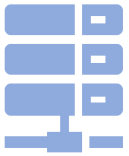
封锁方法

- 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技术
- DBMS通常提供了多种类型的封锁。一个事务对某个数据对象加锁后究竟拥有什么样的控制是由封锁的类型决定的
- 基本封锁类型
  - ✓ 排它锁（eXclusive lock，简记为X锁）
  - ✓ 共享锁（Share lock，简记为S锁）



锁

- 排它锁
  - 排它锁又称为写锁
  - 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
- 共享锁
  - 共享锁又称为读锁
  - 若事务T对数据对象A加上S锁，则其它事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁
  - 这就保证了其他事务在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

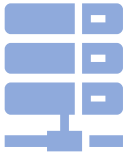


锁

- 锁的相容性矩阵

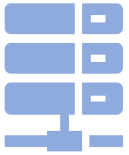
T_2 \ T_1	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

Y=Yes，相容的请求
N=No，不相容的请求



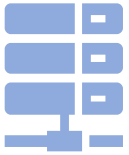
基于锁的并发控制

- 封锁协议（Lock protocol）
  - 基于锁这个工具，设计一组规则，来合理安排一组并发事务的交错运行的指令，使得产生的调度是可串行的和可恢复的，甚至是无级联的
  - 协议内容：
    - ✓ 何时申请X锁或S锁
    - ✓ 持锁时间、何时释放
  - 不同的封锁协议，可以在不同的程度上为并发操作的正确调度提供一定的保证
    - 三级封锁协议
    - 两阶段锁协议



三级封锁协议

- 三级封锁协议
  - 设定封锁时机的规则
    - 何时申请X锁或S锁
    - 持锁时间、何时释放
  - 解决并发操作带来的三种不一致性问题（回顾前边内容）
  - 在不同程度上保证数据的一致性



三级封锁协议

- 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）
  - 1级封锁协议可防止丢失修改
  - 在1级封锁协议中，如果是读数据，不需要加锁的，所以它不能保证可重复读和不读“脏”数据。

三级封锁协议

T_1	T_2
① Xlock A	
获得	
② 读A=16	
	Xlock A
③A←A-1	等待
写回A=15	等待
Commit	等待
Unlock A	等待
④	获得Xlock A
	读A=15
	A←A-1
⑤	写回A=14
	Commit
	Unlock A

一级封锁协议：

没有丢失修改

三级封锁协议

T_1	T_2
① Xlock A 获得	
② 读A=16 A←A-1 写回A=15	
③	读A=15
④ Rollback Unlock A	

一级封锁协议：

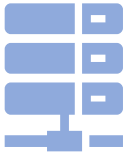
不能阻止读“脏”数据

三级封锁协议

T_1	T_2
①读A=50	
读B=100	
求和=150	
②	Xlock B
	获得
	读B=100
	B←B*2
	写回B=200
	Commit
	Unlock B
③读A=50	
读B=200	
求和=250	
(验算不对)	

一级封锁协议：

不能避免不可重复读



三级封锁协议

- 二级封锁协议
  - 在1级封锁协议基础上，要求事务T在读取数据R前必须先加S锁，读完后即可释放S锁
    - ✓ 2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。

三级封锁协议

T_1	T_2
① Xlock A 获得	
② 读A=16 A←A-1 写回A=15	
③	SlockA 等待
④ Rollback Unlock A	等待 等待 读A=16

二级封锁协议：

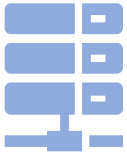
避免读“脏”数据

三级封锁协议

T_1	T_2
① Sclock A	
获得	
读A=50	
Unlock A	
② Sclock B	
获得	Xlock B
读B=100	等待
Unlock B	等待
③ 求和=150	获得Xlock B
	读B=100
	B←B*2
	写回B=200
	Commit
	Unlock B

T_1 (续)	T_2
④Sclock A	
获得	
读A=50	
Unlock A	
Sclock B	
获得	
读B=200	
Unlock B	
求和=250	
(验算不对)	

二级封锁协议： 不可重复读



三级封锁协议

- 三级封锁协议
  - 在1级封锁协议基础上，要求事务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
  - 3级封锁协议可防止丢失修改、读脏数据和不可重复读。

三级封锁协议

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

三级封锁协议：

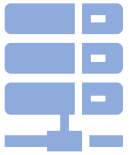
可重复读

三级封锁协议

T_1	T_2
① Xlock C	
读C= 100	
C←C*2	
写回C=200	
②	
	Slock C
	等待
③ ROLLBACK	等待
(C恢复为100)	等待
Unlock C	等待
④	获得Slock C
	读C=100
⑤	Commit C
	Unlock C

三级封锁协议：

不读“脏”数据



三级封锁协议

- 三级封锁协议与事务隔离级别
  - 数据库标准对事务处理的性能和一致性进行了平衡，定义了4种隔离级别
  - 数据库允许用户根据业务需要自行选择隔离级别

1、读未提交(Read Uncommited)

当两个事务A、B同时进行时，即使A事务没有提交，所做的修改也会对B事务内的查询产生影响。对数据进行修改时，会加上共享锁。

2、读已提交(Read commited)

只有在事务提交后，才会对另一个事务产生影响。 **【多数数据库默认的隔离级别】**

3、可重复读（Repeatable Read)

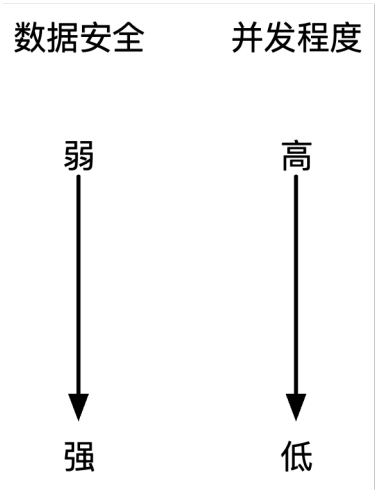
当两个事务同时进行时，其中一个事务修改数据对另一个事务不会造成影响，即使修改的事务已经提交也不会对另一个事务造成影响。

4、串行化（SERIALIZABLE)

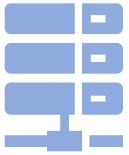
两个事务同时进行时，一个事务读取的数据也会被锁定，不能被别的事务修改

三级封锁协议

- 三级封锁协议与事务隔离级别
  - 数据库允许用户选择隔离级别
  - 不同隔离级别对应于不同等级的封锁协议

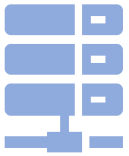


	丢失修改	脏读	不可重复读	幻影读
读未提交	不允许	允许	允许	允许
读已提交	不允许	不允许	允许	允许
可重复读	不允许	不允许	不允许	允许
串行化	不允许	不允许	不允许	不允许



两阶段封锁协议

- 三级封锁协议讨论了封锁的时机，目的是从不同程度上解决数据不一致性问题，但没有讨论调度的正确性
- 为了保证并行操作的正确性，DBMS的并行控制机制必须提供一定的手段来保证调度是可串行化的
- 两阶段锁协议（Two-Phase Locking，简称2PL）是一种能够保证并发执行结果正确性的封锁协议
- 两阶段锁协议内容
  - 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，事务不再获得任何其他封锁



两阶段封锁协议

- “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；
    - ✓ 第二阶段是释放封锁，也称为收缩阶段。

例：

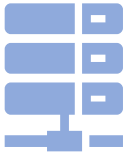
事务1的封锁序列：

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。



两阶段封锁协议

- 若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。

对一组遵守两段锁协议的事务，可以实现可串行化调度，使得其并行执行的结果一定是正确的

（可以用数学归纳法证明：若n-1个事务的2PL是可串行化的，则n个事务的2PL也是可串行化的）

此外还可证明，2PL的可串行化调度等价于按照锁定点顺序执行的串行调度（锁定点Lock Points，指一个事务获得最后一个锁的时间点）

- 另：事务遵守两段锁协议是可串行化调度的充分条件，但不是必要条件。即，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

两阶段封锁协议

- 两段锁协议可以解决所有数据不一致问题

丢失更新问题

时间	事务T_1	数据库中R的值	事务T_2
t_0	SLOCK A	16	
t_1	Read A		
t_2	XLOCK A		SLOCK A
t_3	A=A-1		Wait
t_4	UPDATE A		WAIT
t_5	UNLOCK X	15	READ A
t_6			XLOCK A
t_7			A=A-2
T_8			UPDATE A
t_9		13	UNLOCK X

两阶段封锁协议

- 两段锁协议可以解决所有数据不一致问题

不可重复读问题

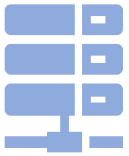
时间	事务T_1	数据库中R的值	事务T_2
t_0		1000	
t_1	SLOCK R		
t_2	Read R		
t_3			XLOCK R
t_4	COMMIT		WAIT
t_5	UNLOCK S		WAIT
t_6			XLOCK R
t_7			Read R
T_8			R=R-200
t_9			UPDATE R
t_{10}			UNLOCK X

两阶段封锁协议

- 两段锁协议可以解决所有数据不一致问题

脏读问题

时间	事务T_1	数据库中R的值	事务T_2
t_0	XLOCK R	1000	
t_1	Read R		
t_2	R=R-100		
t_3	UPDATE R		
t_4		900	SLOCK R
t_5	ROLLBACK		WAIT
t_6	UNLOCK R	1000	SLOCK R
t_7			Read R
T_8			UNLOCK S



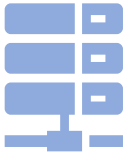
两阶段封锁协议

- 两段锁协议不能解决级联回滚问题

  例：T1的回滚会导致T2的级联回滚

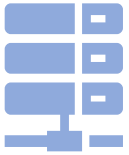
T1:	R(A), W(A), R(B),		W(B), Abort
T2:		R(A), W(A)	

- 解决方案：严格的两段锁协议（Strict Two-phase Locking）
  - 在两阶段锁协议基础上，增加规则：
    - 事务获得的锁只有在事务结束时候才释放
  - 第三级封锁协议与严格的两阶段锁协议一致
    - 遵循第三级封锁协议的事务必然遵守两阶段锁协议



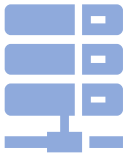
事务分别遵循什么协议？结果如何?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	
	Read(A)
	Unlock(A)
	Lock_S(B)
Lock_X(B)	
	Read(B)
	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	



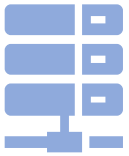
事务分别遵循什么协议？结果如何?

Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	Unlock(A)
rollback	Read(B)
	Unlock(B)
	PRINT(A+B)



事务分别遵循什么协议？结果如何?

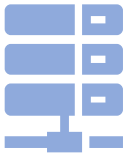
Lock_X(A)	
Read(A)	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
rollback	
Unlock(A)	
Unlock(B)	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)



死锁

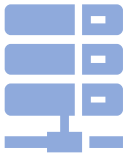
- 两段锁协议不能解决死锁问题

Lock_X(A)	
	Lock_S(B)
	Read(B)
	Lock_S(A)
Read(A)	**wait**
A: = A-50	
Write(A)	
Lock_X(B)	
wait	



死锁

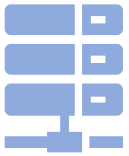
- 解决死锁的方法
  - 1. 预防死锁
  - 2. 死锁的诊断与解除



死锁

1. 预防死锁

- 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
- 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：
    - ✓ 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



死锁

2. 死锁的诊断与解除

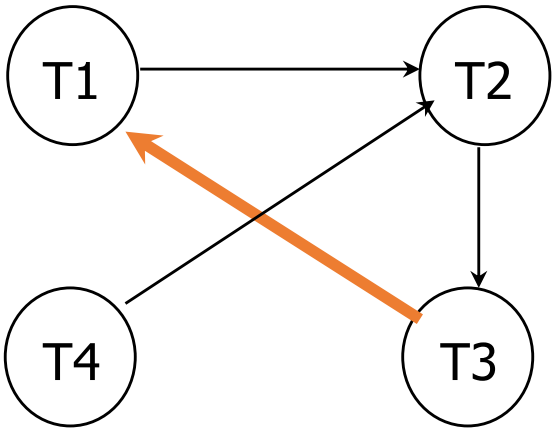
- 允许死锁发生
- 解除死锁
  - ✓ 由DBMS的并发控制子系统定期检测系统中是否存在死锁
  - ✓ 一旦检测到死锁，就要设法解除
- 等待图法检测死锁

死锁

2. 死锁的诊断与解除

- 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T1等待T2，则T1，T2之间划一条有向边，从T1指向T2
- 系统进行周期性检查，如果发现图中存在回路，则表示出现了死锁，为此选择一个事务回滚，打破死锁

T1: S(A), S(D), S(B)
T2: X(B) X(C)
T3: S(D), S(C), X(A)
T4: X(B)



课堂练习

- 有如下三个调度：
  - ✓S1 = $r_3(B), w_3(C), r_1(A), w_1(D), w_2(A), r_2(C), w_4(A), c_2, c_4, r_3(D), c_3, c_1$
  - ✓S2 = $r_3(C), w_1(A), r_2(B), w_3(B), c_3, w_1(C), c_1, r_2(A), c_2$
  - ✓S3 = $w_3(E), w_4(A), w_1(D), r_1(B), w_1(C), c_1, r_2(D), r_4(D), w_3(B), w_4(C), c_4, c_3, c_2$

哪些调度是冲突可串行的？哪些是可恢复的？

课堂练习

- 有如下调度：

r1(X) w1(X) r2(X) r1(Y) w1(Y) r2(Y) w2(Y)

请问是正确的调度吗？若是，等价于什么样的串行调度？

是否是一个2PL的调度？

这是个2PL的调度，加上锁操作后如下 (L指lock，U指unlock) ：

L1(X) L1(Y) r1(X) w1(X) U1(X) L2(X) r2(X) r1(Y) w1(Y) U1(Y) L2(Y) r2(Y) w2(Y)U2(Y)

请问是严格两阶段协议的调度吗？

作业

- 课本第305页第1， 2题
- 课本第326页第1， 2， 3， 4， 6， 7， 8， 9， 10， 12，13，14题
- 课堂练习的两道题
- 提交时间：下次上课之前