



实现篇

第七章 事务处理与恢复

主讲：王金宝

海量数据计算研究中心





- 7.1 事务并发控制
- 7.2 日志与故障恢复





7.1 事务并发控制

- 事务概念
- 事务的并发执行和调度
- 并发控制协议





7.1.1 事务概念

- **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
- **事务与程序不同**
 - 在关系数据库中，一个事务可以是一条SQL语句，一组SQL语句或整个程序
 - 一个程序通常包含多个事务
- **事务是并发控制和恢复的基本单位**





7.1.1 事务概念

• 定义事务

— 显式定义方式

BEGIN TRANSACTION

SQL 语句1

SQL 语句2

。 。 。 。 。

COMMIT

BEGIN TRANSACTION

SQL 语句1

SQL 语句2

。 。 。 。 。

ROLLBACK

- 事务异常终止
- 事务运行的过程中发生了故障，不能继续执行
- 系统将事务中对数据库的所有已完成的操作全部撤销
- 事务回滚到开始时的状态

读+更新)

更新写回到磁盘





7.1.1 事务概念

- 事务的特性 (ACID)

- 原子性 (Atomicity)：即事务完全执行或完全不执行
- 一致性 (Consistency)：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态
 - 一致性状态：
 - 数据库中只包含成功事务提交的结果
 - 不一致性状态：
 - 数据库系统运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态





一致性与原子性



银行转帐：从帐号A中取出一万元，存入帐号B。

— 定义一个事务，该事务包括两个操作

A	B
$A=A-1$	$B=B+1$

— 这两个操作要么全做，要么全不做

- 全做或者全不做，数据库都处于一致性状态
- 如果只做一个操作，用户逻辑上就会发生错误，少了一万元，数据库就处于不一致性状态





7.1.1 事务概念

• 事务的特性 (ACID)

- 隔离性(Isolation): 表面看起来, 每个事务都是在没有其它事务同时执行的情况下执行的
 - 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
 - 并发执行的各个事务之间不能互相干扰
- 持久性(Durability): 一个事务一旦提交, 它对数据库中数据的改变就应该是永久性的
 - 接下来的其他操作或故障不应该对其执行结果有任何影响





并发控制保证I; 故障恢复保证AD; AID保证C

- 保证事务ACID特性是事务处理的任务

- 破坏事务ACID特性的因素

- 多个事务并行运行时，不同事务的操作交叉执行

- 数据库管理系统必须保证多个事务的交叉运行不影响这些事务的隔离性（保证I）

- 事务在运行过程中被强行停止

- 数据库管理系统必须保证被强行终止的事务对数据库和其他事务没有任何影响（保证A和D）

T_1	T_2
Read(A)	
$A := A + 50$	
Write(A)	
	Read(A)
	$A := A * 2$
	Write(A)
	Read(B)
	$B := B * 2$
	Write(B)
Read(B)	
$B := B + 50$	
Write(B)	

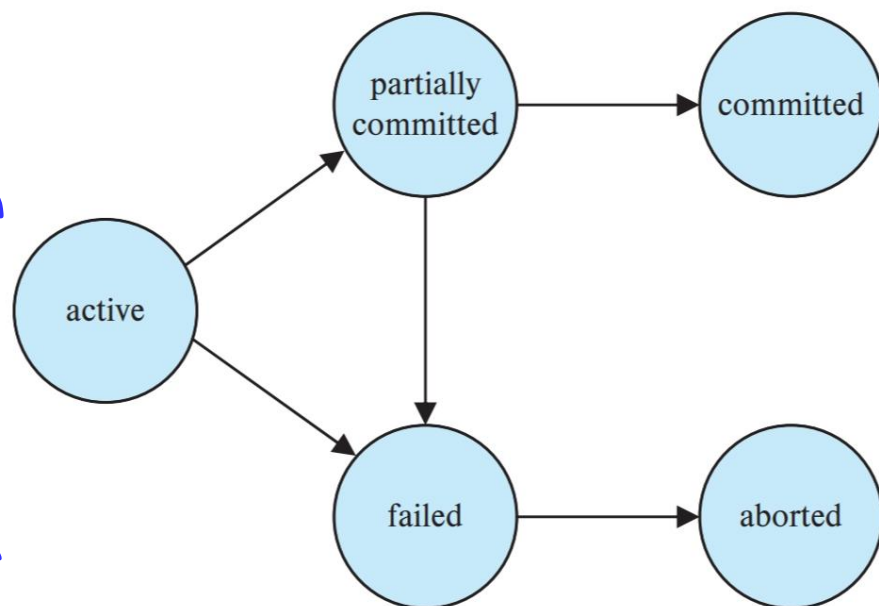




7.1 事务概念

• 事务的状态转移

- Active: 事务正在执行
- Partially committed: 事务执行完最后一条语句
- Committed: 事务成功完成
- Failed: 事务无法继续执行
- Aborted: 事务回滚, 数据库恢复到事务执行之前的状态





READ(X, t)和WRITE(t, X)由事务发出
INPUT(X)和OUTPUT(X)由缓冲区管理器发出

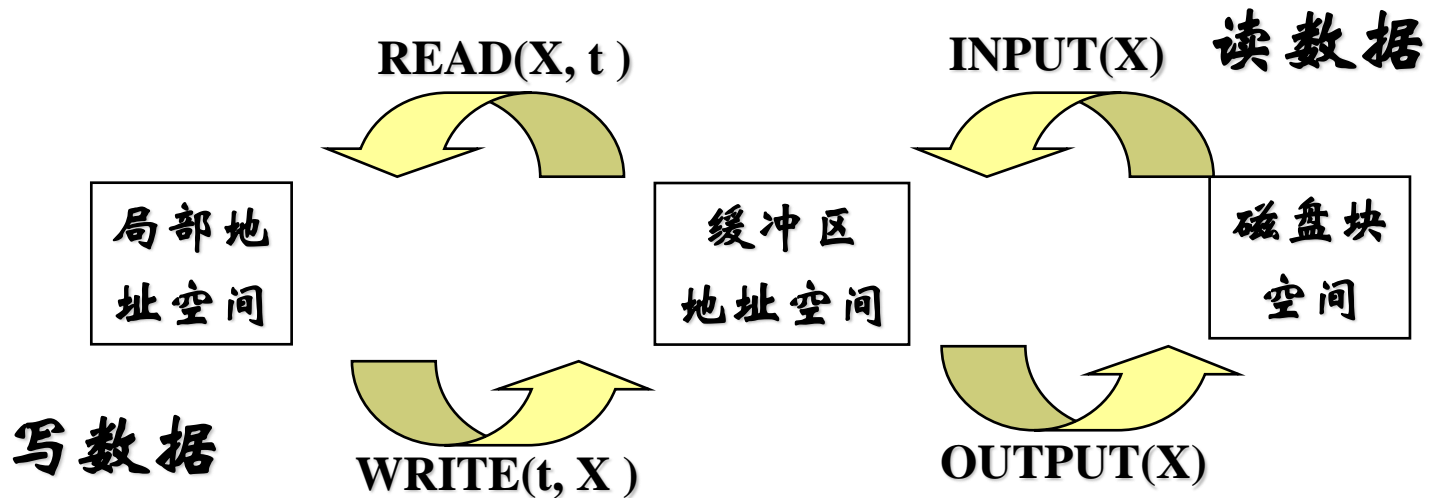
- 事务调用以下两个操作访问数据

- Read(X):

- 从数据库把数据项X传送到事务的局部缓冲区

- Write(X)

- 从事务的局部缓冲区把数据项X传回数据库





7.1.2 事务的并发执行和调度

- 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大量的事务



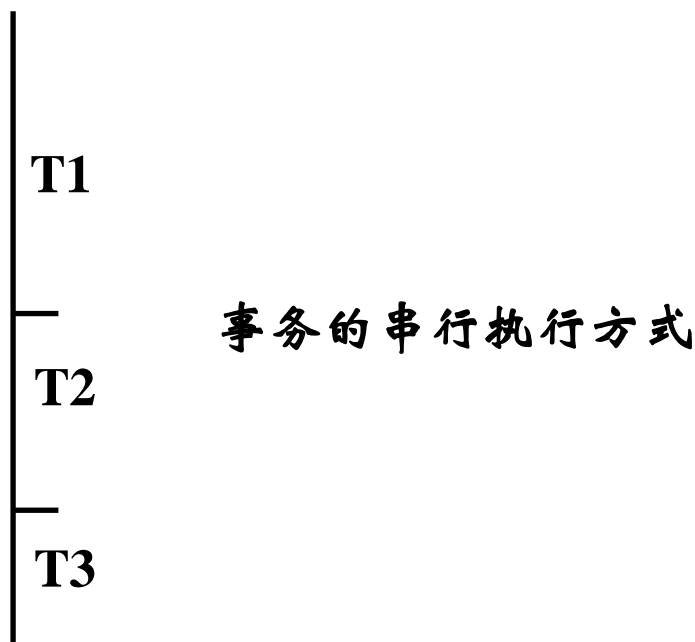


7.1.2 事务的并发执行和调度

• 事务执行的方式

— 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
- 不能充分利用系统资源，发挥数据库共享资源的特点



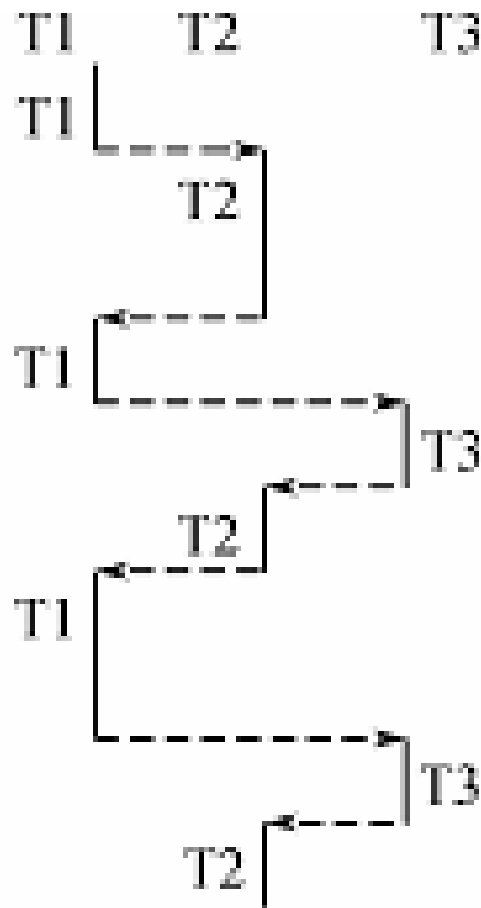


7.1.2 事务的并发执行和调度

• 事务执行的方式

— 交叉并发方式

- 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



事务的交叉并发执行方式





7.1.2 事务的并发执行和调度

- 事务执行的方式

- 同时并发方式

- 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

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





- 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，进而破坏数据库的一致性
- DBMS 必须提供并发控制机制
- 并发控制机制是衡量一个DBMS性能的重要标志之一





并发操作容易带来数据的不一致性，例如：

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: 乙售票点也卖出一张机票，修改余额 $A \leftarrow A-1$ ，所以A为15，把A写回数据库

乙事务修改A并写回后覆盖了甲事务的修改，

甲事务的修改就被丢失

结果明明卖出两张机票，数据库中机票余额只减少1





- 并发操作带来的数据不一致性问题

- 丢失修改 (Lost Update)

- 不可重复读 (Non-repeatable Read)

- 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

- 例如





不可重复读举例：

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

- T1 读取 $B=100$ 进行运算
- T2 读取同一数据 B，对其进行修改后将 $B=200$ 写回数据库
- T1 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





不可重复读举例：

- 事务T1按一定条件从数据库中读取了某些数据记录后，**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
- 事务T1按一定条件从数据库中读取某些数据记录后，**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。

这两种不可重复读有时也称为**幻影**现象（Phantom Row）





• 并发操作带来的数据不一致性问题

— 读“脏”数据 (Dirty Read)

- 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





读“脏”数据举例

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

- 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





7.1.2 事务的并发执行和调度

- 事务的调度

- 一个或多个事务的重要操作按时间排序的一个序列
- READ, WRITE序列

- 串行调度

- 如果一个调度S的动作组成首先是一个事务的所有动作, 然后是另一个事务的所有动作, 依此类推、没有动作的混合, 那么我们说这一调度是串行的

- 更精确地讲, 如果有任意两个事务T和T', 若T的某个动作在T'的某个动作前, 则T的所有动作在T'的所有动作前, 那么调度S是串行的





Let T_1 transfer \$50 from A to B , and
 T_2 transfer 10% of the balance from A to B .

串行调度

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

串行调度是正确的

然而串行调度使系统资源的利用率不高

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

read(A)
 $A := A - 50$
write(A)
read(B)
 $B := B + 50$
write(B)





与串行调度1等价的一个并发调度

提高并发性!!!

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$
<div>!!!</div> $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$	$\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$





一个违反一致性($A=B$)
的并发调度

T_1	T_2
Read(A)	
$A := A+50$	
Write(A)	
	Read(A)
	$A := A*2$
	Write(A)
	Read(B)
	$B := B+50$
	Write(B)

如何保证一个并发调度是正确的??





7.1.2 事务的并发执行和调度

- 可串行化调度

- 如果不管数据库初始状态如何，一个调度对数据库状态的影响都和某个串行调度相同，则我们说这个调度是**可串行化的**

- 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





T ₁	T ₂	A	B
		25	25
Read(A, t)			
t:= t+100			
Write(A, t)		125	
	Read(A, s)		
	s:= s×2		
	Write(A, s)	250	
Read(B, t)			
t:= t+125			
Write(B, t)			150
	Read(B, s)		
	s:= s×2		
	Write(B, s)		300

一个非串行的可串行化调度



T ₁	T ₂	A	B
		25	25
Read(A, t)			
t:= t+100			
Write(A, t)		125	
	Read(A, s)		
	s:= s×2		
	Write(A, s)	250	
	Read(B, s)		
	s:= s×2		
	Write(B, s)		50
Read(B, t)			
t:= t+125			
Write(B, t)			175

一个非可串行化的调度



7.1.2 事务的并发执行和调度

- 事务和调度的一种记法
 - 只考虑事务的读写操作
 - 用 $r_i(X)$ 、 $w_i(X)$ 表示事务 T_i 读和写数据库元素 X
- $T1 : r1(A), w1(A), r1(B), w1(B)$
 $T2 : r2(A), w2(A), r2(B), w2(B)$
- 事务集合 T 的调度 S 是组成它的事务动作的一个交错的序列

$r1(A), w1(A), r2(A), w2(A), r1(B), w1(B), r2(B), w2(B)$





7.1.2 事务的并发执行和调度

- 冲突

— 调度中一对连续的动作，它们满足：如果它们的顺序交换，那么涉及的事务中至少有一个的行为会改变





例如，设 T_i 、 T_j 是不同的事务，即 $i \neq j$

- $r_i(X); r_j(Y)$ 是不冲突的，即使 $X=Y$ 。原因是这些步骤都不会改变任何值；
- 如果 $X \neq Y$ ，那么 $r_i(X); w_j(Y)$ 不会是冲突的。原因是 T_i 读 X 对 T_j 没有影响，因此它不会影响 T_j 为 Y 写的值；
- 如果 $X \neq Y$ ，那么 $w_i(X); r_j(Y)$ 亦不会是冲突的。
- 只要 $X \neq Y$ ，那么 $w_i(X); w_j(Y)$ 不会是冲突的。





例如，设 T_i 、 T_j 是不同的事务，即 $i \neq j$

- 同一事务的两个动作是冲突的，如 $r_i(X); w_i(Y)$ 。
原因在于单个事务的动作顺序是固定的，不能被DBMS重新排列
- 不同事务对同一数据库元素的写冲突。即 $w_i(X); w_j(X)$ 是一个冲突
- 不同事务对同一数据库元素的读和写也冲突。即 $r_i(X); w_j(X)$ 是冲突的， $w_i(X); r_j(X)$ 也是

不同事务的任何两个动作在顺序上可以交换，除非

- 它们涉及同一数据库元素，并且
- 至少有一个动作是写





7.1.2 事务的并发执行和调度

- 两个调度是冲突等价的
 - 我们说两个调度是冲突等价的，如果通过一系列相邻动作的非冲突交换能将它们中的一个转换为另一个
- 冲突可串行化
 - 如果一个调度冲突等价于一个串行调度。那么我们说该调度是冲突可串行化的

一个调度若是冲突可串行化的，则一定是可串行化的调度





考虑如下调度:

$r1(A), w1(A), r2(A), w2(A), r1(B), w1(B), r2(B), w2(B)$

我们说这个调度是冲突可串行化的

按照定义, 将这一调度转换为串行调度(T_1, T_2)

通过交换相邻动作将冲突可串行化调度转换为串行调度

$r1(A), w1(A), r2(A), w2(A), r1(B), w1(B), r2(B), w2(B)$

$r1(A), w1(A), r2(A), r1(B), w2(A), w1(B), r2(B), w2(B)$

$r1(A), w1(A), r1(B), r2(A), w2(A), w1(B), r2(B), w2(B)$

$r1(A), w1(A), r1(B), r2(A), w1(B), w2(A), r2(B), w2(B)$

$r1(A), w1(A), r1(B), w1(B), r2(A), w2(A), r2(B), w2(B)$

如何更简单地判断一个调度是否是冲突可串行化的?





7.1.2 事务的并发执行和调度

• 优先图

- 已知调度S，其中涉及事务T1和T2，可能还有其它事务。我们说T1优先于T2，记作 $T1 < T2$ ，如果有T1的动作A1和T2的动作A2，满足：
 - 在S中A1在A2前；
 - A1和A2都涉及同一数据库元素；并且
 - A1和A2中至少有一个动作是写。
- 因此，在任何冲突等价于S的调度中，A1将出现在A2前。所以，如果这些调度中有一个是串行调度，那么该调度必然使T1在T2前





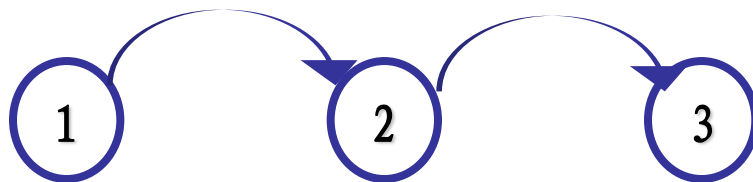
7.1.2 事务的并发执行和调度

- 优先图

- 优先图的结点是调度S中的事务。当这些事务是具有不同的 i 的 T_i 时，我们将仅用整数 i 来表示 T_i 的结点。如果 $T_i < T_j$ ，则有一条从结点 i 到结点 j 的弧

- 例如，有调度S：

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$



调度顺序为： $T_1 \rightarrow T_2 \rightarrow T_3$





7.1.2 事务的并发执行和调度

- 冲突可串行性判断

- 构造调度S的优先图，判断其中是否有环

- 例1，调度S：

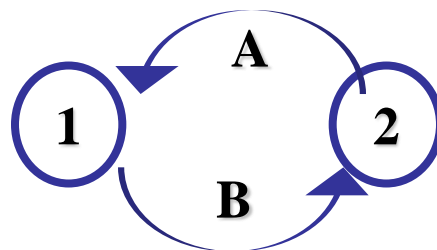
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$



- 例2，考虑调度S'：

$r_2(A); r_1(B); w_2(A); r_2(B); r_1(A); w_1(B); w_1(A); w_2(B);$

对应的优先图，





- 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
- 存在不满足冲突可串行化条件的可串行化调度

例：3个事务 $T1=W1(Y)W1(X)$, $T2=W2(Y)W2(X)$, $T3=W3(X)$

存在两个调度： $L1=W1(Y)W1(X)W2(Y)W2(X)W3(X)$

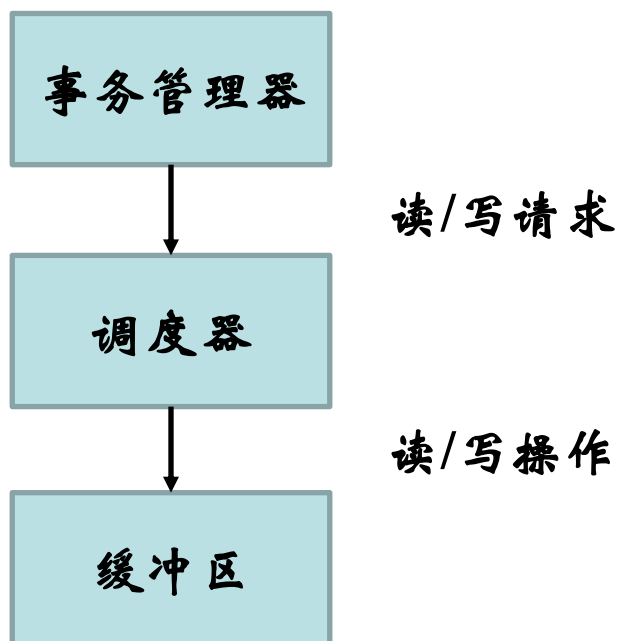
$L2=W1(Y)W2(Y)W2(X)W1(X)W3(X)$

$L1$ 是一个串行调度。

$L2$ 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







基于锁的并发控制协议

- 锁的概念

- 锁是数据库元素上的并发控制标志
- 锁的类型有很多，我们考虑两种类型：

- 共享锁(S):

如果事务 T 得到了数据库元素 Q 上的共享锁 S ，则 T 可以读 Q ，但不能写 Q

- 排他锁(X): 也称互斥锁

如果事务 T 得到了数据库元素 Q 上的排他锁 X ，则 T 既可以读 Q ，也可以写 Q





基于锁的并发控制协议

- 锁的概念

- 多粒度的锁

上锁的数据库元素可以是：

- 关系表：

- 对应表级锁

- 磁盘块(或页)

- 对应块(页)级锁

- 元组

- 对应元组级锁





基于锁的并发控制协议

- 锁的概念

- 每个事务在存取一个数据库元素之前必须获得这个数据库元素上的锁
- 一个事务需要获得的锁的类型依赖于它将在数据库元素上执行什么样的操作





基于锁的并发控制协议

- 锁的概念

- 给定一个各种类型锁的集合，如下定义这个锁集合上的相容关系

- 令 A 和 B 表示任意类型的锁。设事务 T_i 在数据库元素 Q 上要求一个 A 型锁，事务 T_j ($T_i \neq T_j$) 已经在 Q 上有一个 B 型锁。如果事务 T_i 能够获得 Q 上的 A 型锁，则说 A 型锁和 B 型锁是相容的

	共享锁	排他锁
共享锁	是	否
排他锁	否	否





• 锁的概念

- 事务通过执行 $\text{LOCK-S}(Q)$ 操作申请数据库元素 Q 上的共享锁。
- 事务通过执行 $\text{LOCK-X}(Q)$ 操作申请数据库元素 Q 上的排他锁。
- $\text{UNLOCK}(Q)$ 操作用来释放 Q 上的锁。
- 如果事务 T 要存取数据库元素 Q , T 必须申请在 Q 上加锁。如果 Q 已经被其他的事务加以非共享锁, 则事务 T 必须等待, 直到所有其他事务的非共享锁全部被释放。
- 事务 T 可以释放它在任何数据库元素上所加的任何类型的锁。





- 银行数据库系统的例子：

- 设A和B是两个帐号，A和B的初始值分别是100和200元
- 事务 T_1 从帐号B向帐号A转50元钱，事务 T_2 显示帐号A和B的总金额

- T_1 : LOCK-X(B);
 READ(B);
 B := B - 50;
 WRITE(B);
 UNLOCK(B);
 LOCK-X(A);
 READ(A);
 A := A + 50;
 WRITE(A);
 UNLOCK(A)。

- T_2 : LOCK-S(A);
 READ(A);
 UNLOCK(A);
 LOCK-S(B);
 READ(B);
 UNLOCK(B);
 DISPLAY(A+B)。

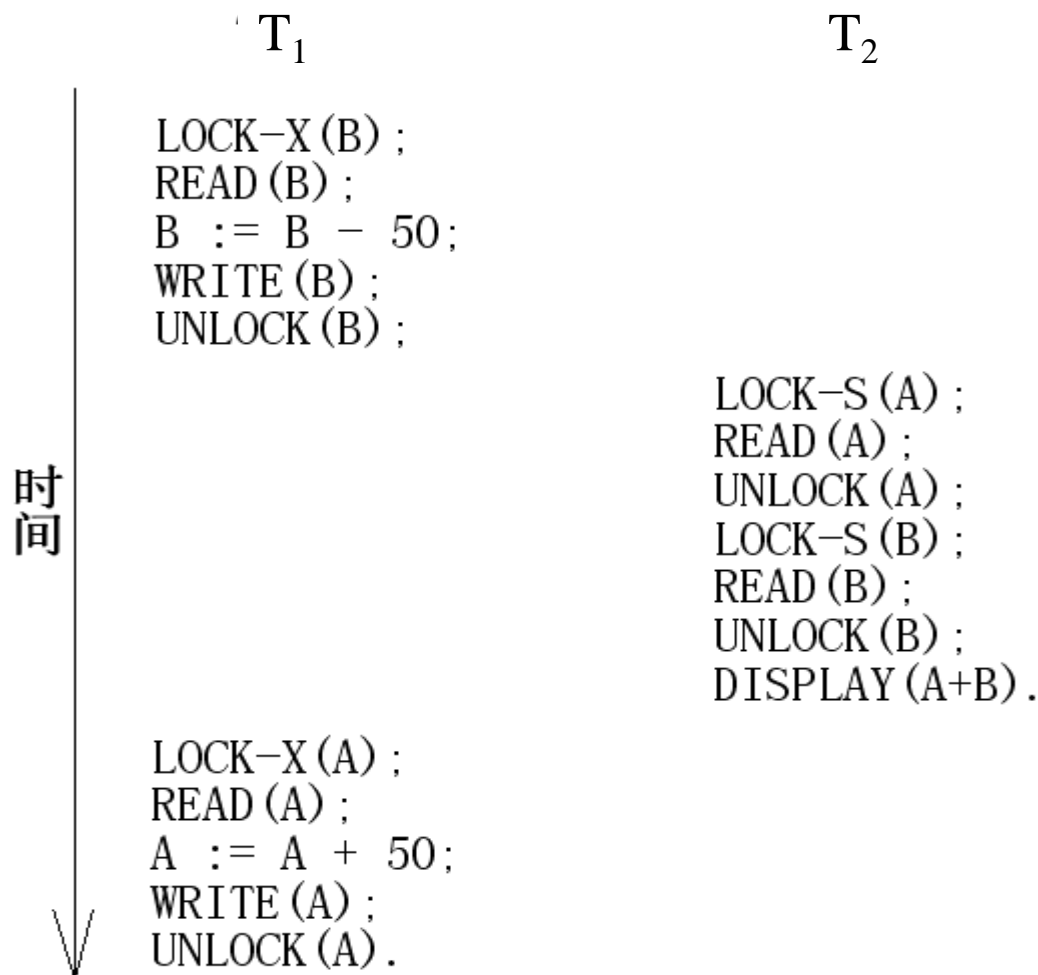
两个事务串行执行，即 $T_1 \rightarrow T_2$ 或 $T_2 \rightarrow T_1$ 方式执行， T_2 总是显示300美元的值





- 一个事务只要存取一个数据库元素，它就必须持有该数据库元素上的一个锁
- 如果一个事务完成了对一个数据库元素的最后一次存取之后就立即放弃它的锁，则不能确保调度的可串行性

事务 T_2 错误地显示250元!!





• 两阶段锁协议

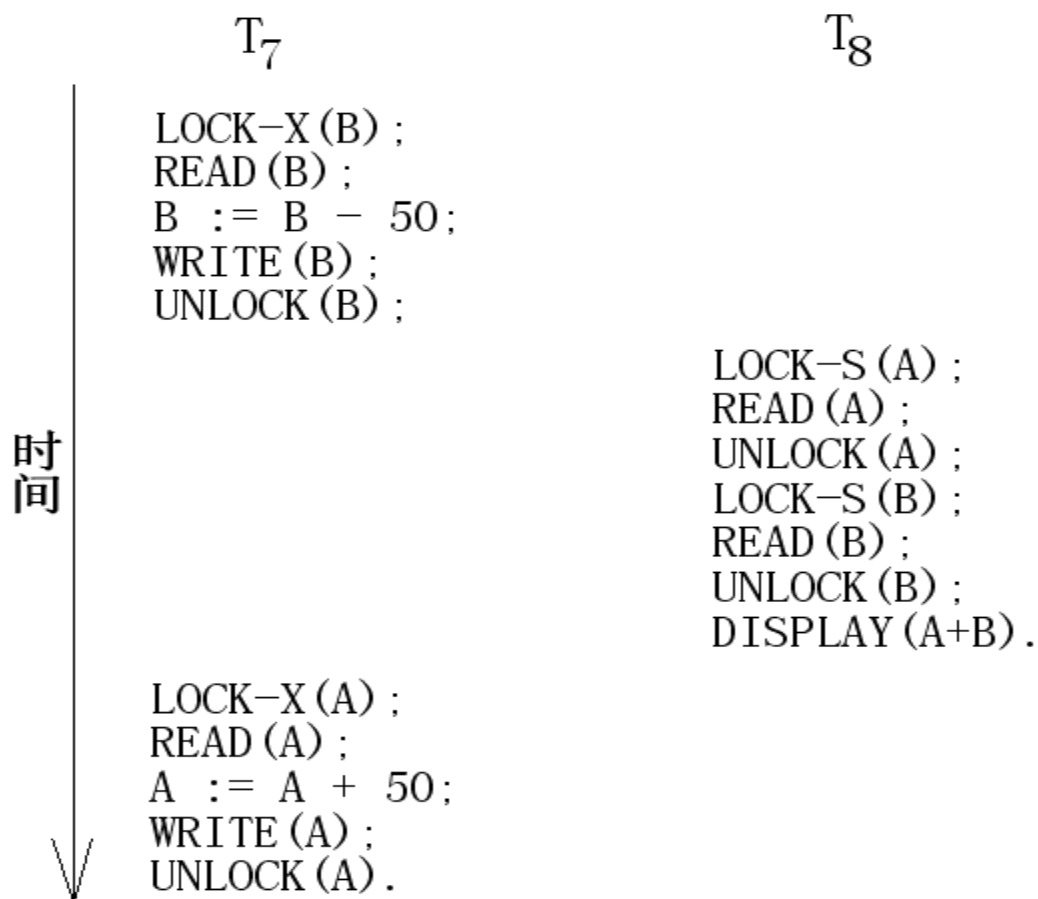
- 两阶段锁协议要求每个事务分两个阶段进行数据库元素的加锁和解锁
 - 阶段1 加锁阶段：在这阶段，事务可以申请获得任何数据库元素上的任何类型的锁，但是**不能释放任何锁**
 - 阶段2 解锁阶段：在这阶段，事务可以释放任何数据库元素上的任何类型的锁，但是**不能再申请任何锁**
- 每个事务开始运行后即进入加锁阶段，申请获得所需要的所有锁
- 当一个事务第一次释放锁时，该事务进入解锁阶段。进入解锁阶段的事务不能再申请任何锁

可以证明：任何一个满足两阶段锁协议的合理调度都是冲突可串行的！





• 满足两段锁协议?



No !





• 满足两段锁协议? Yes!

T_1	T_2	A	B
		25	25
$L_1(A); R_1(A);$			
$A := A+100;$			
$W_1(A); L_1(B); U_1(A);$		125	
	$L_2(A); R_2(A);$		
	$A := A*2;$		
	$W_2(A);$	250	
	$L_2(B)$ 被拒绝;		
$R_1(B); B := B+100;$			
$W_1(B); U_1(B);$			125
	$L_2(B); U_2(A); R_2(B);$		
	$B := B*2;$		
	$W_2(B); U_2(B);$		250





- 满足两段锁协议? Yes!

T3: LOCK-X(B);
READ(B);
B := B - 50;
WRITE(B);
LOCK-X(A);
READ(A);
A := A + 50;
WRITE(A);
UNLOCK(B);
UNLOCK(A)。

T4: LOCK-S(A);
READ(A);
LOCK-S(B);
READ(B);
DISPLY(A+B);
UNLOCK(A);
UNLOCK(B)。

存在的问题?





• 死锁处理

— 在数据库系统运行期间，如果存在一组事务 $\{T_0, T_1, \dots, T_n\}$ ，使得 T_0 等待 T_1 持有的数据库元素锁， T_1 等待 T_2 持有的锁，……， T_{n-1} 等待 T_n 持有的锁， T_n 等待 T_0 持有的锁，则称系统处于死锁状态

— 解决死锁

• 超时处理

• 预防死锁

- 使用预防死锁协议，如：数据库图协议、时间戳协议等
- 数据对象按全局排序，按顺序加锁，可以避免死锁
- 制定允许等待锁的机制：wait-die (旧事务等新事务拥有的锁，新事务等旧事务则新事务被回滚)、wound-wait (新事务等旧事物拥有的锁，旧事物抢新事务拥有的所)

• 死锁检测与恢复





• 死锁检测

— 事务等待图

- 一个有向图 $G(V, E)$, 其中:

- V 是系统中所有事务

- 有序对 $(T_i, T_j) \in E$ 表示 T_i 等待 T_j 释放其所需要的数据库元素

— 死锁检测

- 等待图中是否存在回路

— 什么时刻检测?

- 根据死锁产生的频率

- 根据死锁事务的个数





• 死锁恢复

— 选择撤销的死锁事务，考虑如下因素：

- 事务已经完成的计算量和未完成的计算量
- 事务已经使用过的数据库元素
- 事务还需要使用的数据库元素
- 需要同时撤销的事务数

— 事务的撤销：如何撤销？

- 一个简单方法：全部放弃该事务，及UNDO所有操作，重新启动事务
- 更有效的方法：从后向前UNDO这个事务的操作，直到死锁解除





两段锁协议不能避免级联中止

T1	T2	T3
Lock-X(A)		
Lock-S(B)		
Write(A)		
Unlock(A)		
Unlock(B)	Lock-S(A)	
	Lock-X(C)	
	Read(A)	
	Write(C)	
	Unlock(C)	Lock-S(C)
	Unlock(A)	Lock-X(D)
		Read(C)
		Write(D)
		Unlock(C)
Abort		Unlock(D)





- **strict two-phase locking protocol**
 - 服从两段锁协议
 - 排他锁必须保持持有，直到事务提交
- **rigorous two-phase locking protocol**
 - 服从两段锁协议
 - 所有的锁必须保持持有，直到事务提交





“活锁”现象



- 事务 T_1 封锁了数据R
- 事务 T_2 又请求封锁R，于是 T_2 等待。
- T_3 也请求封锁R，当 T_1 释放了R上的封锁之后系统首先批准了 T_3 的请求， T_2 仍然等待。
- T_4 又请求封锁R，当 T_3 释放了R上的封锁之后系统又批准了 T_4 的请求……
- T_2 有可能永远等待，这就是活锁的情形





—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





关于释放锁的时机选择及其影响的讨论

(与两段锁独立的讨论)

	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		√		√	√	√	√





• 数据库图协议

— 另一种基于锁的并发控制协议

— 数据库图

• 设 DB 是一个数据库， DB 的数据库图是一个有向图 $G(D, E)$ ，其中：

— $D=\{d_1, d_2, \dots, d_n\}$ 是 DB 中所有数据库元素的集合

— E 是边集， $(d_i, d_j) \in E$ 当且仅当所有存取 d_i 和 d_j 的事务都先存取 d_i ，后存取 d_j





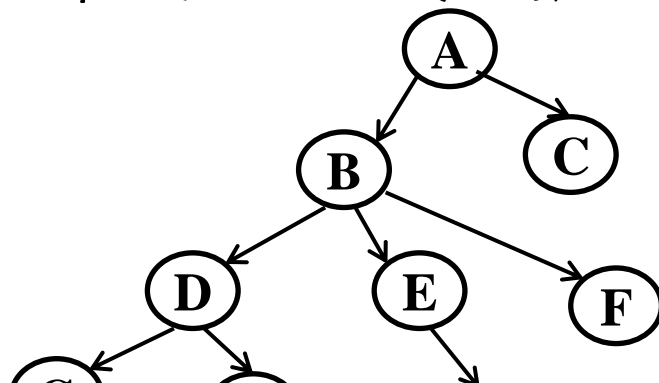
• 数据库图协议

— 一个简单的数据库图协议：树协议

- 仅允许一种锁，即排他锁LOCK-X
- 每个事务T对一个数据库元素只能加一次锁，加锁规则：
 - T的第一个锁可以加到任何数据库元素上
 - 数据库元素Q可以由T加锁，仅当Q的父节点已被T加锁
 - 数据库元素可以在任何时刻被解锁
 - 一个已经被T加锁并解锁的数据库元素不能由T再加锁

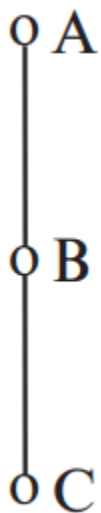
如图所示：

若事务T要存取数据库元素B、C，
则T要先对A加锁，然后再对B、C
加锁



树协议不但可以保证调度是冲突可串行的，
而且保证无死锁！





A, B, C 三个
数据库对象
组成的树

T_1	T_2
lock (A)	
lock (B)	
unlock (A)	
	lock (A)
lock (C)	
unlock (B)	
	lock (B)
	unlock (A)
	unlock (B)
unlock (C)	

T_1	T_2
lock (A)	
	lock (B)
lock (C)	
	unlock (B)
unlock (A)	
unlock (C)	

两段锁允许，但是树协议不
允许的调度

树协议允许，但是两段锁不
允许的调度



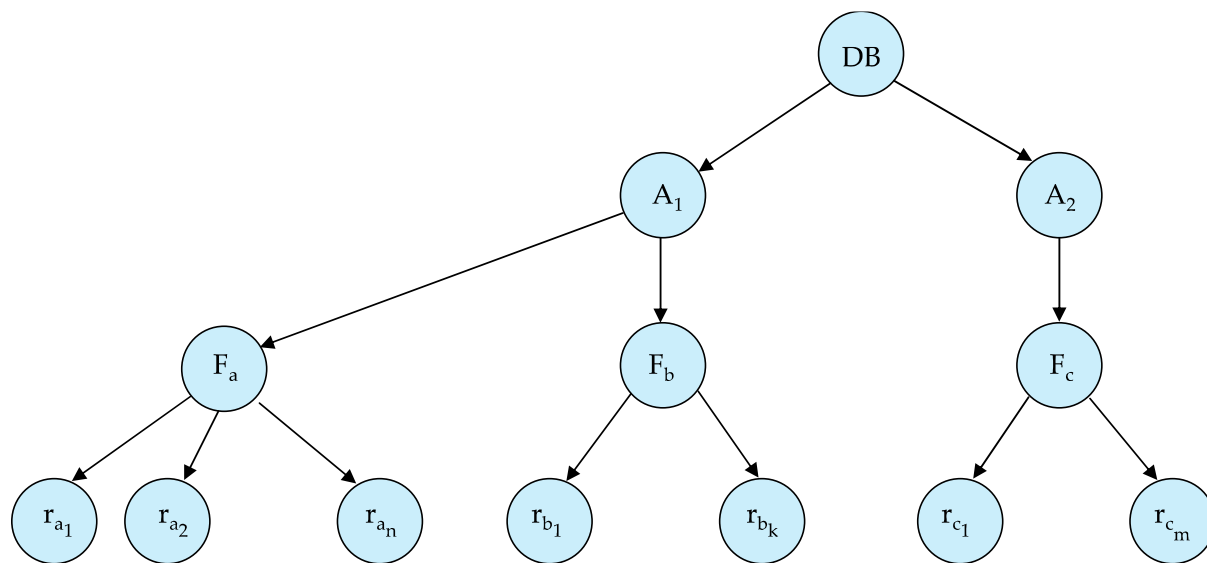
- 多粒度锁

- 允许数据项具有各种大小，并定义数据粒度层次结构，其中小粒度嵌套在较大粒度中
- 数据项根据粒度层次表示成一颗树，注意不要和树协议混淆
- 当事务显式地锁定树中的一个节点时，它会以相同的模式隐式地锁定该节点的所有后代
- 锁定粒度(完成锁定的树层次):
 - 细粒度(树中较低):并发性高，锁管理的开销大
 - 粗粒度(在树中较高):锁定的管理开销小，并发性低



- 粒度层次示例

— 从粗粒度到细粒度的层次，包括数据库、存储区域、文件、元组，还可以包括属性



- 新的锁模式：意向锁

— 除了S锁和X锁，增加三种意向锁：

- IS锁(intention-shared):表示要在该节点的某些子孙节点上加共享锁。
- IX锁(intention-exclusive):表示要在该节点的某些子孙节点上加排他锁或共享锁
- SIX锁(shared and intention-exclusive):表示要在该节点为根的子树上加共享锁，并在某些子孙节点上加排他锁



- 五种锁模式的相容矩阵

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false



多粒度锁

• 多粒度锁协议

—事务 T_i 可以在节点 Q 按照如下规则加锁：

- 遵守锁的相容矩阵
- 根节点必须首先被加锁，可以加任意模式的锁
- 节点 Q 可以被事务 T_i 加S锁或IS锁，仅当 Q 的父节点上有 T_i 加的IX锁或IS锁
- 节点 Q 可以被事务 T_i 加X锁或SIX锁或IX锁，如果仅当 Q 的父节点上有 T_i 加的IX锁或SIX锁
- T_i 可以给节点加锁，仅当 T_i 还没有执行任何解锁，遵循两段锁
- T_i 可以给节点 Q 解锁，仅当 T_i 不持有 Q 的任何子节点上的锁

• 加锁过程从根到叶子，解锁过程从叶子到根

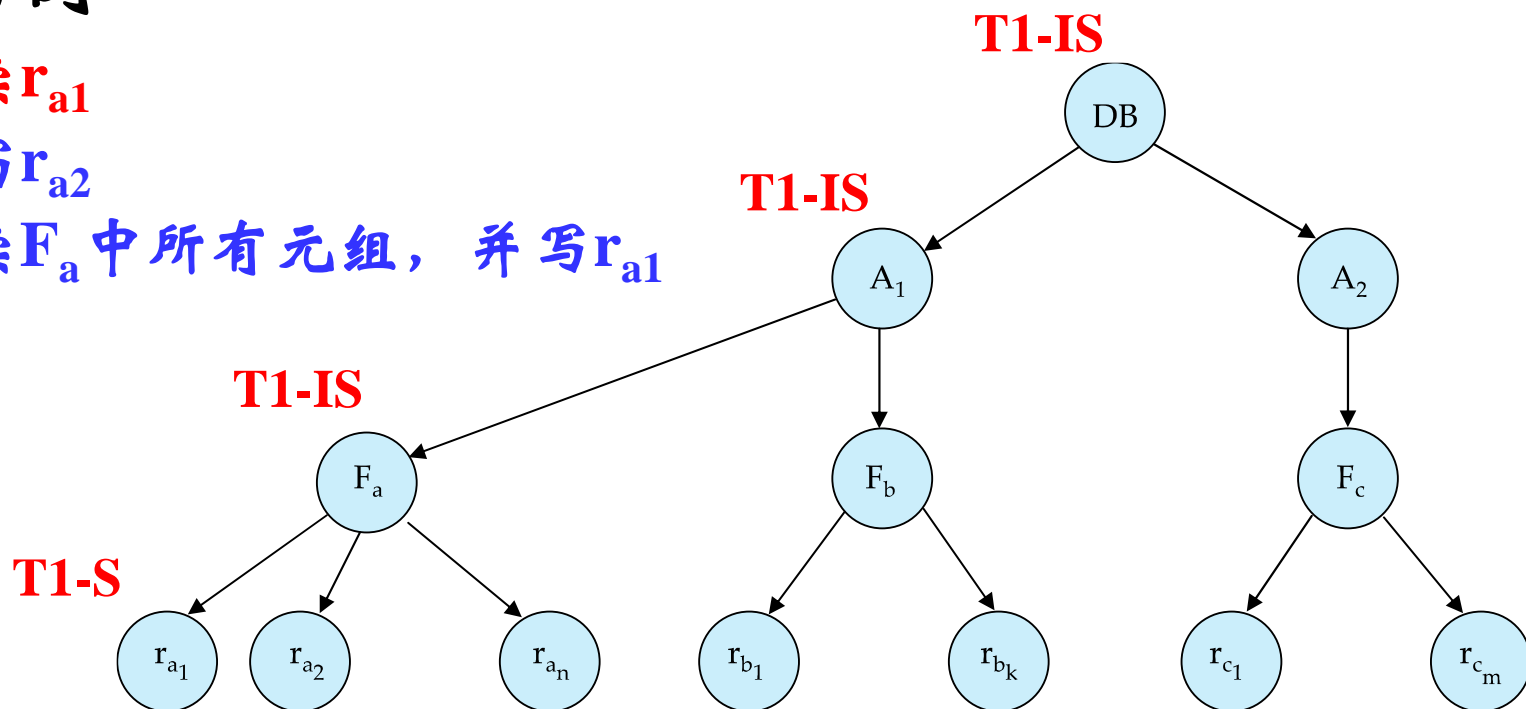


多粒度锁

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

• 加锁示例

- T1: 读 r_{a1}
- T2: 写 r_{a2}
- T3: 读 F_a 中所有元组, 并写 r_{a1}

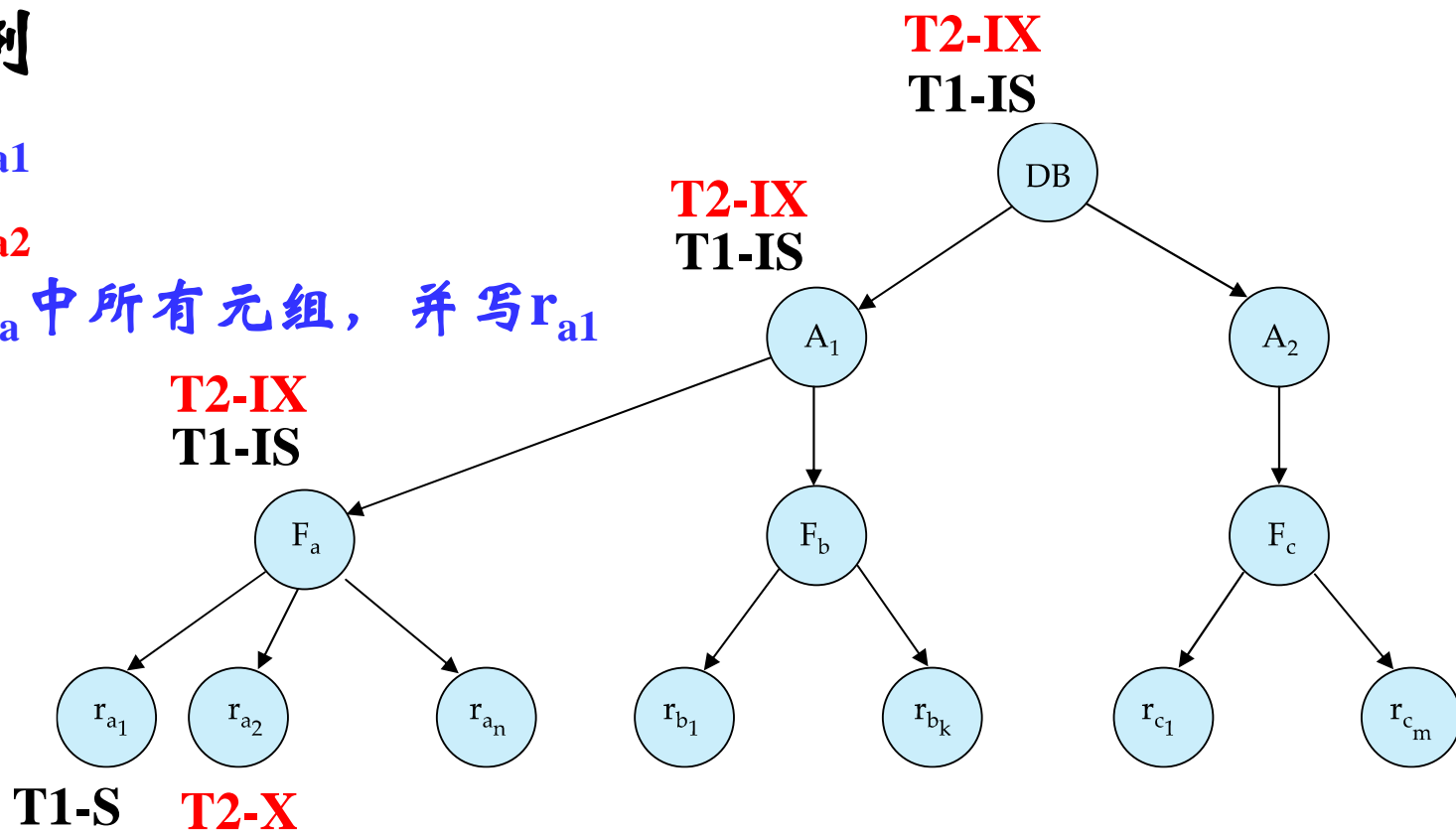


多粒度锁

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

• 加锁示例

- T1: 读 r_{a1}
- T2: 写 r_{a2}
- T3: 读 F_a 中所有元组, 并写 r_{a1}

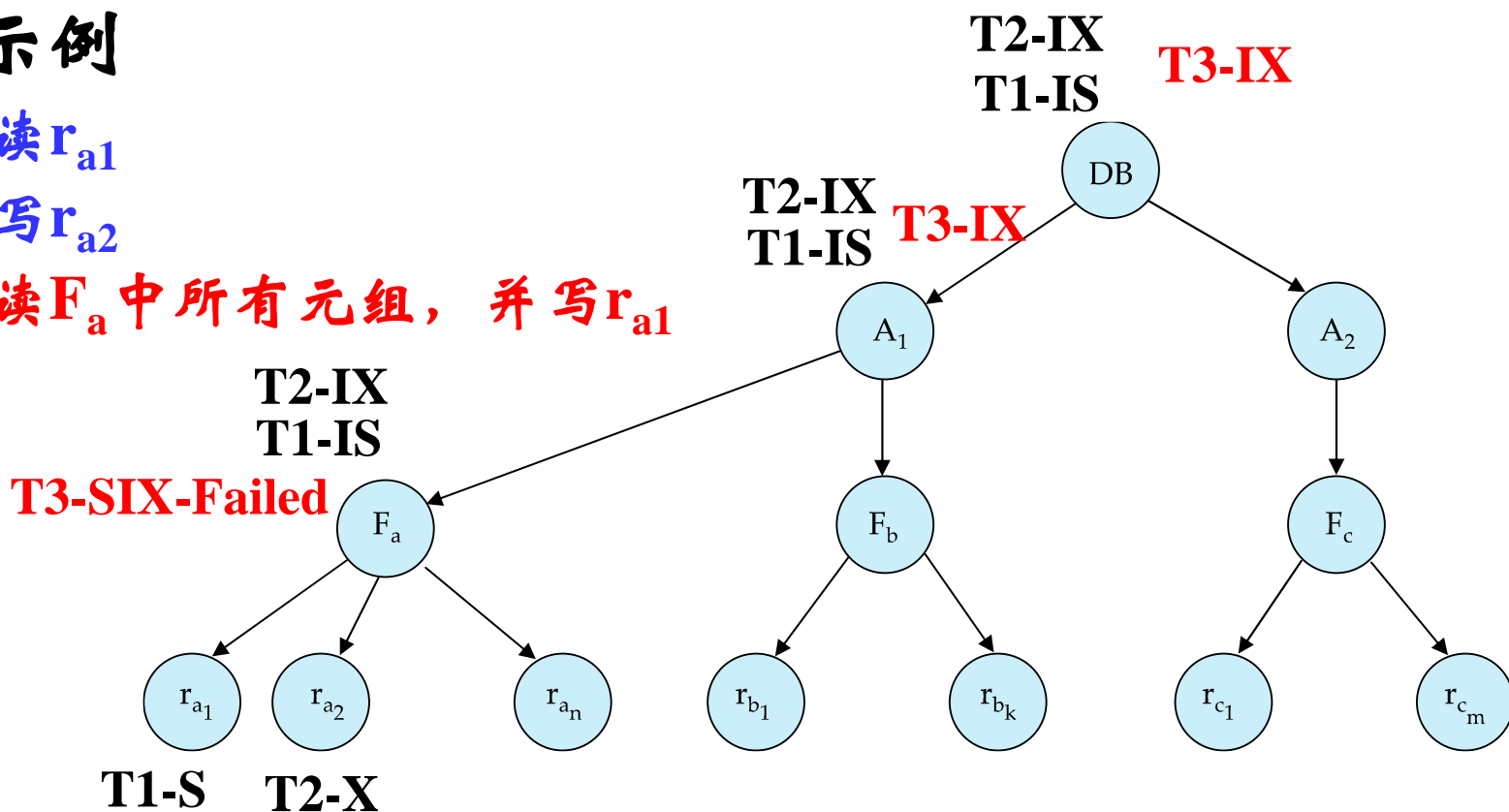


多粒度锁

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

• 加锁示例

- T1: 读 r_{a1}
- T2: 写 r_{a2}
- T3: 读 F_a 中所有元组, 并写 r_{a1}





• 基于时间戳的协议

— 系统中每个事务 T_i ，分配一个固定的时间戳，记为 $TS(T_i)$

- 事务 T_i 开始执行前，由系统分配的
- 若事务 T_i 已被分配时间戳 $TS(T_i)$ ，并且有一新事务 T_j 进入系统，则 $TS(T_i) < TS(T_j)$

— 事务的时间戳决定了串行化顺序，即

- 若 $TS(T_i) < TS(T_j)$ ，则系统产生的调度必须等价于 T_i 出现在 T_j 之前的某个串行调度





• 基于时间戳的协议

- 系统中每个数据库元素 Q ，关联了两个时间戳
 - $W-TS(Q)$: 成功完成 $W(Q)$ 的事务的最大时间戳
 - $R-TS(Q)$: 成功完成 $R(Q)$ 的事务的最大时间戳
- 每当有新的 $W(Q)$ 和 $R(Q)$ 执行时，更新 $W-TS(Q)$ 和 $R-TS(Q)$
- 对于事务 T 的读写请求，调度器可能的决定
 - 同意请求
 - 中止 T ，回滚并重启 T





时间戳协议不但可以保证调度是冲突可串行的，而且保证无死锁！但不能避免级联中止。

• 基于时间戳的协议

— 若事务 T_i 执行 $R(Q)$

- 如果 $TS(T_i) < W-TS(Q)$ ，则 T_i 需读入的 Q 值已经被覆盖， $R(Q)$ 被拒绝，同时事务 T_i 回滚
- 如果 $TS(T_i) > W-TS(Q)$ ，则执行 $R(Q)$ ，且

$$R-TS(Q) = \max\{R-TS(Q), TS(T_i)\}$$

— 若事务 T_i 执行 $W(Q)$

- 如果 $TS(T_i) < R-TS(Q)$ ，则 T_i 需写入的 Q 值已经被其后新的事务读出， $W(Q)$ 被拒绝，同时事务 T_i 回滚
- 否则，如果 $TS(T_i) < W-TS(Q)$ ，则 T_i 要写入的 Q 值已经过时，这个 $W(Q)$ 可以被忽略
- 否则，执行 $W(Q)$ ，且 $W-TS(Q) = TS(T_i)$





• 基于时间戳的协议

— 问题：

- 当一系列冲突的短事务引起长事务反复重启时，可能导致长事务**饿死现象**

— 解决办法：

- 当发现一个事务反复重启时，让发生冲突的事务暂时停止执行，过一段时间后再重启该事务

时间戳和基于锁的协议

基于锁的协议中，事务等待锁时通常被延迟；

高冲突情况下，时间戳协议造成大量的事务回滚，严重影响性能；

冲突少时，时间戳性能更好；

冲突多时，基于锁的协议性能更好。



事务隔离等级

- 常用的DBMS提供了四种事务的隔离等级
 - 可串行化、可重复读、读已提交、读未提交
 - Global (对整个系统生效), Session (仅对提交事务的连接生效)

	脏写	脏读	不可重复读	幻象
读未提交	No	Yes	Yes	Yes
读已提交	No	No	Yes	Yes
可重复读	No	No	No	Yes
可串行化	No	No	No	No





- 7.1 并发控制
- 7.2 日志与故障恢复





7.2.1 故障及种类

- 故障是不可避免的
 - 计算机硬件故障
 - 软件的错误
 - 操作员的失误
 - 恶意的破坏
- 故障的影响
 - 运行事务非正常中断，影响数据库中数据的正确性
 - 破坏数据库，全部或部分丢失数据





7.2.1 故障及种类

- 数据库恢复

数据库管理系统必须具有把数据库从错误状态恢复到某一已知的正确状态(亦称为一致状态或完整状态)的功能, 这就是数据库的恢复管理系统对故障的对策

- 恢复子系统是数据库管理系统的一个重要组成部分
- 恢复技术是衡量系统优劣的重要指标





7.2.1 故障及种类

- 故障种类

- 介质故障(硬故障)

- 外存故障

- 系统故障(软故障)

- 是指造成系统停止运转的任何事件，使得，
- 系统要重新启动

- 事务故障

- 事务内部引发

- 计算机病毒

- 一种人为的故障或破坏，是一些恶作剧者研制的一种计算机程序
- 可以繁殖和传播，造成对计算机系统包括数据库的危害





介质故障

- 导致原因

磁盘损坏、磁头碰撞、瞬时强磁场干扰等

- 介质故障破坏数据库或部分数据库，并影响正在存取这部分数据的所有事务

- 介质故障恢复

- 装入数据库发生介质故障前某个时刻的数据副本
- 重做自此时开始的所有成功事务，将这些事务已提交的结果重新记入数据库
- 磁盘重大故障：利用RAID





• 故障表现

- 整个系统的正常运行突然被破坏
- 所有正在运行的事务都非正常终止
- 不破坏数据库
- 内存中数据库缓冲区的信息全部丢失

• 常见原因

- 特定类型的硬件错误（如CPU故障）
- 操作系统故障
- DBMS代码错误
- 系统断电





• 系统故障恢复

- 发生系统故障时，一些尚未完成的事务的结果可能已送入物理数据库，造成数据库可能处于不正确状态
 - 恢复策略：系统重新启动时，恢复程序要强行撤消(Undo)所有未完成事务
- 发生系统故障时，有些已完成的事务可能有一部分甚至全部留在缓冲区，尚未写回到磁盘上的物理数据库中，系统故障使得这些事务对数据库的修改部分或全部丢失
 - 恢复策略：系统重新启动时，恢复程序需要重做(Redo)所有已提交的事务





事务故障

- 事务故障分类

- 可预期的事务故障

- 可以通过事务程序本身发现

- 更多的是非预期的，不能由应用程序处理

- 运算溢出
 - 并发事务发生死锁而被选中撤销该事务
 - 违反了某些完整性限制等，如：输入某一位数是错的

以后，事务故障仅指这类非预期的故障





事务故障

- 事务故障意味着
 - 事务没有达到预期的终点(COMMIT或者显式的ROLLBACK)
 - 数据库可能处于不正确状态
- 事务故障的恢复: **撤销事务(Undo)**
 - 强行回滚 (ROLLBACK) 该事务
 - 撤销该事务已经作出的任何对数据库的修改, 使得该事务象根本没有启动过一样





计算机病毒

- 计算机病毒已成为计算机系统的主要威胁，自然也是数据库系统的主要威胁
- 数据库一旦被破坏仍要用恢复技术把数据库加以恢复
 - 不仅涉及数据库技术





数据库恢复技术

- 数据库恢复涉及的关键问题
 - 如何建立冗余数据
 - 数据转储 (backup)
 - 日志文件 (logging)
 - 如何利用这些冗余数据实施数据库恢复





7.2.2 基于日志的恢复

- 日志文件

是日志记录的一个序列，用于记载数据库事务对数据库的更新操作情况

- 日志记录的格式

- $\langle \text{Start } T_i \rangle$ ，表示事务 T_i 已经开始
- $\langle \text{Commit } T_i \rangle$ ，表示事务 T_i 成功完成
- $\langle \text{Abort } T_i \rangle$ ，事务 T_i 未成功，被中止
- $\langle T_i, X, V_{old}, V_{new} \rangle$ ，表示事务 T_i 对数据库元素 X 执行了更新操作，其中：
 - V_{old} 是 X 的原值，若更新操作为插入，则 V_{old} 为空
 - V_{new} 是 X 的新值，若更新操作为删除，则 V_{new} 为空





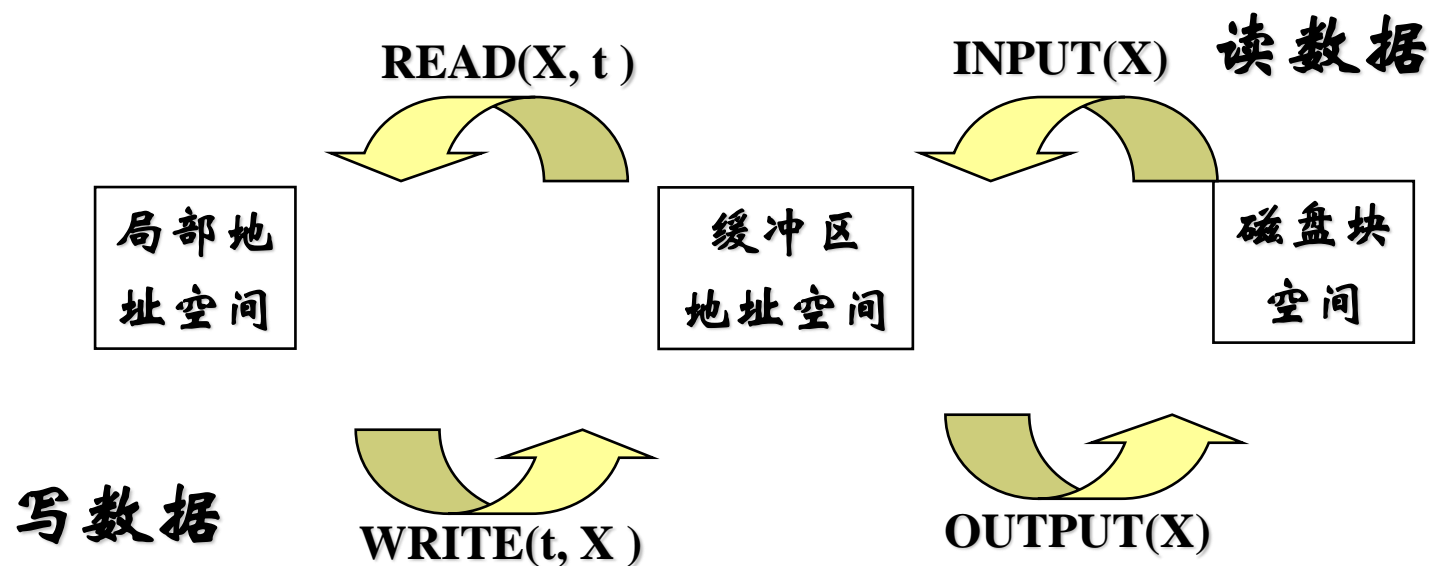
7.2.2 基于日志的恢复

- 日志用途
 - 进行事务故障恢复
 - 进行系统故障恢复
 - 协助后备副本进行介质故障恢复
- 为保证数据库是可恢复的，日志文件必须遵循两条原则
 - 次序严格按并行事务执行的时间次序
 - 必须先写日志文件，后写数据库
 - Write Ahead Logging (WAL)





• 事务执行读/写数据的过程



READ(X, t)和WRITE(t, X)由事务发出
INPUT(X)和OUTPUT(X)由缓冲区管理器发出





缓冲技术

- 日志缓冲技术
- 数据库缓冲技术





- 日志缓冲技术

- 通常一个日志记录远远小于永久存储器的读写单位
- 经常向永久存储器写单个日志记录将导致很大的系统开销
- 对日志记录的读写操作使用缓冲技术
 - 主存中设立缓冲区，其大小等于永久存储器的读写单位
 - 被写的日志记录先存储到缓冲区，缓冲区满之后再一起写入永久存储器

日志记录最初在主存中创建，一旦合适，
日志记录就被永久地写到磁盘上

Flash Log可以将日志强制刷新到磁盘





- 缓冲区中的日志记录在系统发生故障时会丢失
- 为了保证事务的原子性，数据库恢复协议需遵守如下的规则：
 - 任何事务 T 必须在日志记录 $\langle \text{COMMIT } T \rangle$ 已经写入永久存储器后才可以进入提交状态
 - 任何事务 T 的非 $\langle \text{COMMIT } T \rangle$ 型日志记录必须在日志记录 $\langle \text{COMMIT } T \rangle$ 写入永久存储器之前写入永久存储器
 - 主存缓冲区中的数据库数据必须在所有与这些数据有关的日志记录被写入永久存储器之后才可以写入永久存储器的数据库中





- 数据库缓冲技术

- 当系统要把数据块B2读入主存并覆盖数据块B1时，它必须按如下方式进行

- IF B1已经被修改
 - THEN
 - 输出有关B1中数据的所有日志记录到永久存储器
 - 输出B1到磁盘
 - 从磁盘输入B2到内存缓冲区
 - ENDIF





7.2.2 基于日志的恢复

- 基于日志的数据库恢复技术
 - 推迟更新技术
 - 即时更新技术





推迟更新技术

- 推迟更新协议

- 每个事务在提交之前不能更新数据库(Output)
- 在一个事务所有更新操作对应的日志记录写入永久存储器之前，该事务不能提交

- 也就是：

对于任一事务 T_i ，按照如下顺序向磁盘输出 T_i 的日志信息及数据更新信息：

- 首先，执行 $\text{Output}\langle T_i, X, V_{old}, V_{new} \rangle$
- 其次， $\text{Output}\langle \text{COMMIT } T_i \rangle$ 日志记录
- 最后，执行 $\text{OUTPUT}(X)$ ，将更新数据信息写到磁盘





- 每个事务在到达提交点之前不能更新数据库
- 在一个事务的所有更新操作对应的日志记录写入永久存储器之前，该事务不能到达提交点

步骤	动作	t	$M-A$	$M-B$	$D-A$	$D-B$	日志
1					8	8	< START T>
2	Read(A, t)	8	8		8	8	
3	$t := t \times 2$	16	8		8	8	
4	Write(A, t)	16	16		8	8	< T, A, 8, 16>
5	Read(B, t)	8	16	8	8	8	
6	$t := t \times 2$	16	16	8	8	8	
7	Write(B, t)	16	16	16	8	8	< T, B, 8, 16>
8	Flush Log						
9							< COMMIT T>
10	Flush Log						
11	Output(A)	16	16	16	16	8	
12	Output(B)	16	16	16	16	16	





• 如何利用日志记录进行恢复

— 从后往前开始扫描日志，建立两个事务表

- 提交事务表：已经提交的，且磁盘上有日志记录 $\langle \text{COMMIT } T \rangle$ 的所有事务
- 未提交事务表：那些具有 $\langle \text{START } T \rangle$ 日志记录，但无对应 $\langle \text{COMMIT } T \rangle$ 的所有事务





• 如何利用日志记录进行恢复

— 对于提交事务表中的每个事务 T ，执行Redo(T):


- 对于 T 对应的每个更新记录 $\langle T, X, V_{old}, V_{new} \rangle$ ，将数据库中 X 的值改为 V_{new}

— 对于未提交事务表中的事务 T ，执行Undo(T):

- 从日志中删除 T 的信息，放弃 T
- 待以后重新启动 T 执行

**undo和redo操作必须是幂等的，
即执行多次和执行一次的效果相同**






步骤	动作	t	$M-A$	$M-B$	$D-A$	$D-B$	日志
1					8	8	< START T>
2	Read(A, t)	8	8		8	8	
3	$t := t \times 2$	16	8		8	8	
4	Write(A, t)	16	16		8	8	< T, A, 8, 16>
5	Read(B, t)	8	16	8	8	8	
6	$t := t \times 2$	16	16	8	8	8	
7	Write(B, t)	16	16	16	8	8	< T, B, 8, 16>
8	Flush Log						
9							< COMMIT T>
10	Flush Log						
11	Output(A)	16	16	16	16	8	
12	Output(B)	16	16	16	16	16	

— 若执行到第2~9步中的任意一步发生故障，磁盘日志文件中没有事务 T 对应的< COMMIT T >，则

- 从日志中删除 T 的信息，放弃 T ，待以后重新启动运行





步骤	动作	t	$M-A$	$M-B$	$D-A$	$D-B$	日志
1					8	8	< START T >
2	Read(A, t)	8	8		8	8	
3	$t := t \times 2$	16	8		8	8	
4	Write(A, t)	16	16		8	8	< T, A, 8, 16 >
5	Read(B, t)	8	16	8	8	8	
6	$t := t \times 2$	16	16	8	8	8	
7	Write(B, t)	16	16	16	8	8	< T, B, 8, 16 >
8	Flush Log						
9							< COMMIT T >
10	Flush Log						
11	Output(A)	16	16	16	16	8	
12	Output(B)	16	16	16	16	16	

- 若执行完第10步后发生故障，则表示 T 已经完全执行完
- 利用日志，redo事务 T ，修改磁盘上A、B的值为16



有如下两个事务 T_1 、 T_2 ： 初始 $A=1000$, $B=300$

T_1 : Read(A);
 $A:=A-50$;
 Write(A);
 Read(B);
 $B:=B+50$;
 Write(B)

T_2 : Read(A);
 $A:=A-100$;
 Write(A);
 Read(B);
 $B:=B+100$;
 Write(B)

考虑如下调度：

$r1(A)$, $w1(A)$, $r2(A)$, $w2(A)$, $r1(B)$, $w1(B)$, $r2(B)$, $w2(B)$

问：基于推迟更新策略，

向磁盘输出日志文件和更新数据信息的过程？

步骤	动作	t_1	t_2	$M-A$	$M-B$	$D-A$	$D-B$	日志
1						1000	300	< START T_1 >
2	$R1(A, t)$	1000		1000		1000	300	
3	$t := t-50$	950		1000		1000	300	
4	$W1(A, t)$	950		950		1000	300	< $T_1, A, 1000, 950$ >
5	$R2(A, t)$		950	950		1000	300	< START T_2 >
6	$t := t-100$		850	950		1000	300	
7	$W2(A, t)$		850	850		1000	300	< $T_2, A, 950, 850$ >
8	$R1(B, t)$	300			300	1000	300	
9	$t := t+50$	350			300	1000	300	
10	$W1(B, t)$	350			350	1000	300	< $T_1, B, 300, 350$ >
11	Flush Log							
12								< COMMIT T_1 >
13	Flush Log							
14	Output(A)			850		850	300	
15	Output(B)				350	850	350	
16	$R2(B, t)$		350		350	850	350	
17	$t := t+100$		450		350	850	350	
18	$W2(B, t)$		450		450	850	350	< $T_2, B, 350, 450$ >
19	Flush Log							
20								< COMMIT T_2 >
21	Flush Log							
22	Output(B)				450	850	450	



即时更新技术

- 即时更新
 - 允许事务直接更新数据库
- 处于活动状态的事务直接在数据库上实施的更新称为非提交更新
- 即时更新协议
 - 在一个事务的所有更新操作对应的日志记录写入永久存储器之前，事务不能更新数据库
 - 在一个事务的所有更新操作对应的日志记录写入永久存储器之前，事务不允许提交



- 在一个事务的所有更新操作对应的日志记录写入永久存储器之前，事务不能更新数据库
- 在一个事务的所有更新操作对应的日志记录写入永久存储器之前，事务不允许提交

步骤	动作	t	$M-A$	$M-B$	$D-A$	$D-B$	日志
1							< START T>
2	Read(A, t)	8	8		8	8	
3	$t := t \times 2$	16	8		8	8	
4	Write(A, t)	16	16		8	8	< T, A, 8, 16>
5	Flush Log	16	16		8	8	
6	Output(A)	16	16		16	8	
7	Read(B, t)	8	16	8	16	8	
8	$t := t \times 2$	16	16	8	16	8	
9	Write(B, t)	16	16	16	16	8	< T, B, 8, 16>
10	Flush Log						
11	Output(B)	16	16	16	16	16	
12							< COMMIT T>





• 如何利用日志记录进行恢复

— 从尾部开始扫描日志，建立两个事务表

- 提交事务表：已提交，且磁盘上有日志记录 $\langle \text{COMMIT } T \rangle$ 的所有事务
- 未提交事务表：那些具有 $\langle \text{START } T \rangle$ 日志记录，但无对应 $\langle \text{COMMIT } T \rangle$ 的所有事务





• 如何利用日志记录进行恢复

— 对于未提交事务表中的事务 T ，执行Undo(T):


- 对于 T 对应的每个更新记录 $\langle T, X, V_{old}, V_{new} \rangle$ ，将数据库中 X 的值改为 V_{old}
- 从日志中删除 T 的信息

— 对于提交事务表中的每个事务 T ，执行Redo(T):

- 对于 T 对应的每个更新记录 $\langle T, X, V_{old}, V_{new} \rangle$ ，将数据库中 X 的值改为 V_{new}

**undo和redo操作必须是幂等的，
即执行多次和执行一次的效果相同**





步骤	动作	t	$M-A$	$M-B$	$D-A$	$D-B$	日志
1							< START T >
2	Read(A, t)	8	8		8	8	
3	$t := t \times 2$	16	8		8	8	
4	Write(A, t)	16	16		8	8	< T, A, 8, 16 >
5	Read(B, t)	8	16	8	8	8	
6	$t := t \times 2$	16	16	8	8	8	
7	Write(B, t)	16	16	16	8	8	< T, B, 8, 16 >
8	Flush Log						
9	Output(A)	16	16	16	16	8	
10	Output(B)	16	16	16	16	16	
11							< COMMIT T >
12	Flush Log						

— 若执行到第2~11步中的任意一步发生故障

- 利用日志，执行Undo

— 若执行完第12步后发生故障，则表示 T 已经完全执行完

- 利用日志，执行Redo



有如下两个事务 T_1 、 T_2 ： 初始 $A=1000$, $B=300$

T_1 : Read(A);
 $A:=A-50$;
 Write(A);
 Read(B);
 $B:=B+50$;
 Write(B)

T_2 : Read(A);
 $A:=A-100$;
 Write(A);
 Read(B);
 $B:=B+100$;
 Write(B)

考虑如下调度：

$r1(A)$, $w1(A)$, $r2(A)$, $w2(A)$, $r1(B)$, $w1(B)$, $r2(B)$, $w2(B)$

问：基于即时更新策略，

向磁盘输出日志文件和更新数据信息的过程？

步骤	动作	t_0	t_1	$M-A$	$M-B$	$D-A$	$D-B$	日志
1						1000	300	< START T_1 >
2	$R1(A, t)$	1000		1000		1000	300	
3	$t := t-50$	950		1000		1000	300	
4	$W1(A, t)$	950		950		1000	300	< $T_1, A, 1000, 950$ >
5	Flush Log							
6	Output(A)			950		950	300	
7	$R2(A, t)$		950	950		950	300	< START T_2 >
8	$t := t-100$		850	950		950	300	
9	$W2(A, t)$		850	850		950	300	< $T_2, A, 950, 850$ >
10	Flush Log							
11	Output(A)			850		850	300	
12	$R1(B, t)$	300			300	850	300	
13	$t := t+50$	350			300	850	300	
14	$W1(B, t)$	350			350	850	300	< $T_1, B, 300, 350$ >
15	Flush Log							
16	Output(B)				350	850	350	
17								< COMMIT T_1 >
18	$R2(B, t)$		350		350	850	350	
19	$t := t+100$		450		350	850	350	
20	$W2(B, t)$		450		450	850	350	< $T_2, B, 350, 450$ >
21	Flush Log							
22	Output(B)				450	850	450	
23								< COMMIT T_2 >



具有检查点的恢复

- 当系统故障发生后，数据库恢复机制必须搜索日志，确定哪些事务需要redo，哪些事务需要undo
- 如果检查所有日志记录，有两个问题
 - 搜索整个日志将耗费大量的时间
 - 很多需要redo处理的事务已经将它们的操作结果写到数据库中了。重新执行这些操作将浪费大量时间
- 解决办法：具有检查点的恢复方法





具有检查点的恢复

- 具有检查点的恢复方法

- 系统在执行期间动态地维护日志文件，并且在日志中增加一类新的记录<checkpoint L>

- L为建立检查点时刻，当前系统中所有活跃的事务集合

- 数据库恢复机制定期地执行如下操作

- 将目前主存中的所有日志记录输出到永久存储器
 - 将所有缓冲中被修改的数据库的数据块写入磁盘
 - 把日志记录<checkpoint>写入永久存储器

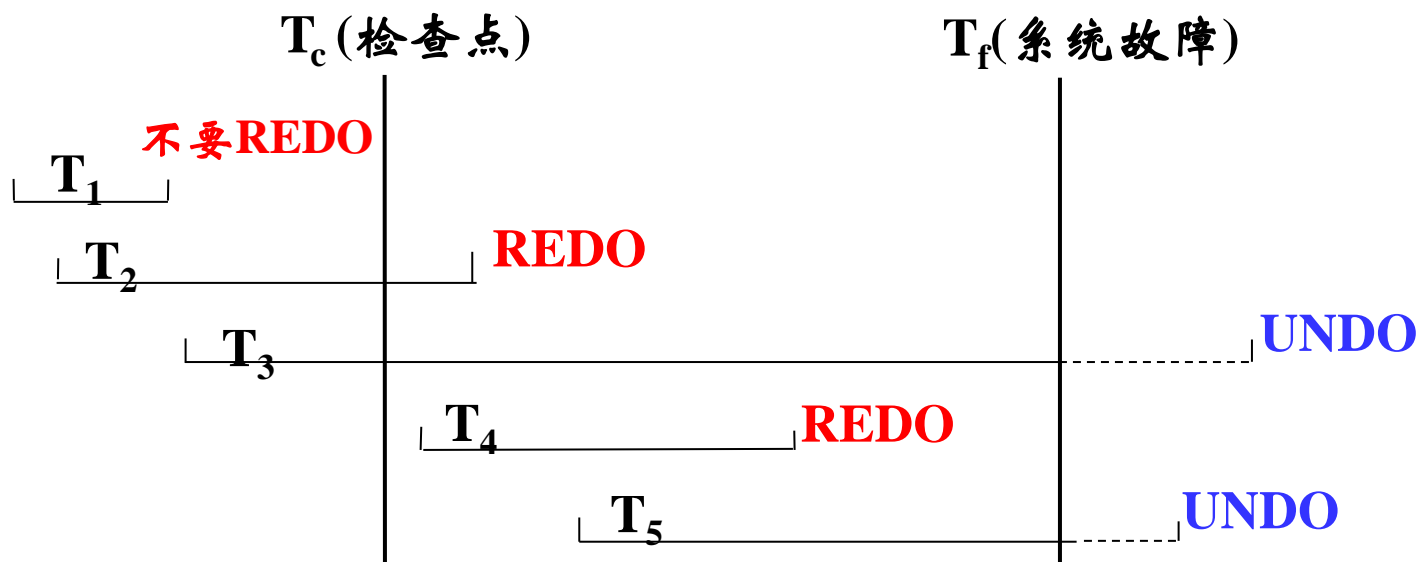
检查点执行过程中不允许事务执行任何更新动作，
如：写缓冲区或写日志记录





具有检查点的恢复

- 使用检查点方法可以改善恢复效率
 - 当事务 T 在一个检查点之前提交
 - T 对数据库所做的修改已写入数据库中
 - 在进行恢复处理时，没有必要对事务 T 执行REDO操作





具有检查点的恢复

- 使用检查点恢复策略
 - 由后向前搜索日志文件，找到第一个检查点记录 $\langle \text{Checkpoint } L \rangle$ ，得到建立检查点时刻系统中活跃的事务集合 L
 - 将 L 中事务放入 Undo-List
 - 从检查点开始正向扫描日志文件，直到日志文件结束
 - 如有新开始的事务 T_i ，把 T_i 暂时放入 Undo-List 队列
 - 如有提交的事务 T_j ，把 T_j 从 Undo-List 移到 Redo-List
 - 对 Undo-List 中的每个事务执行 Undo 操作
 - 对 Redo-List 中的每个事务执行 Redo 操作





数据库镜像

- 介质故障是对系统影响最为严重的一种故障，严重影响数据库的可用性
 - 介质故障恢复比较费时
 - 为预防介质故障，DBA必须周期性地转储数据库
- 提高数据库可用性的解决方案
 - 数据库镜像 (Mirror)





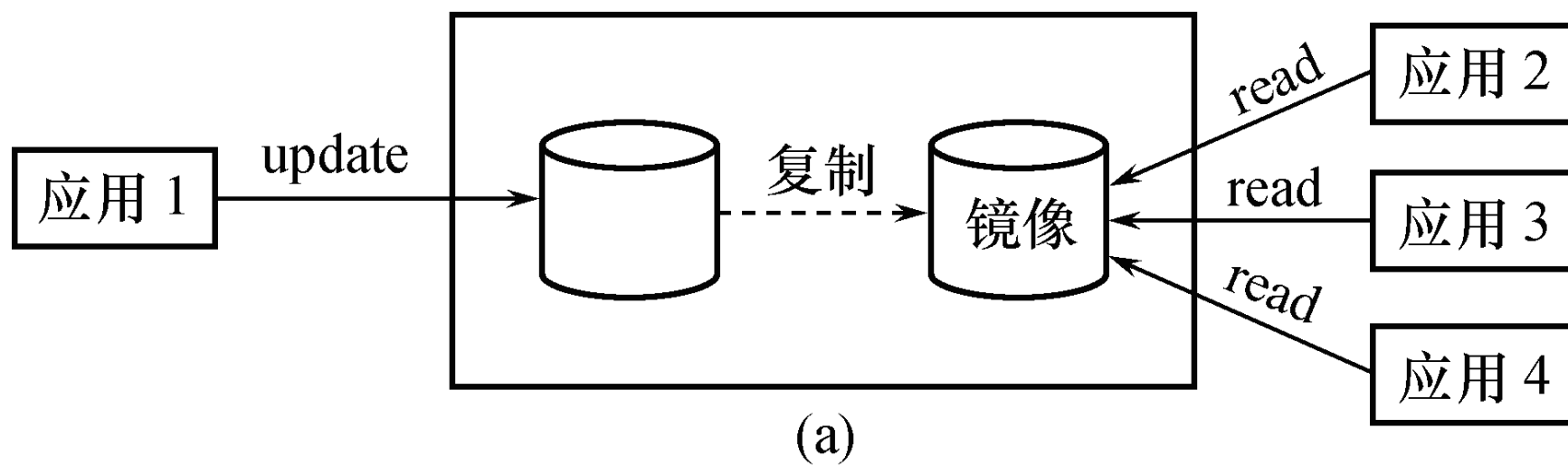
数据库镜像

- 数据库镜像

- DBMS自动把整个数据库或其中的关键数据复制到另一个磁盘上
- DBMS自动保证镜像数据与主数据的一致性

每当主数据库更新时，DBMS自动把更新后的数据复制过去



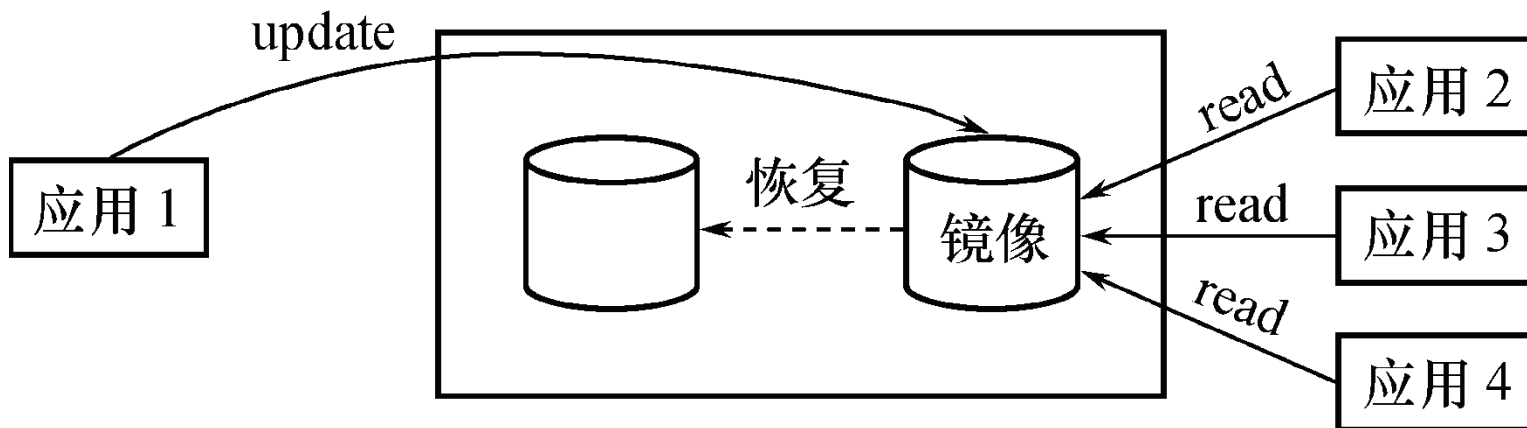




数据库镜像

- 出现介质故障时

- 可由镜像磁盘继续提供使用
- 同时DBMS自动利用镜像磁盘数据进行数据库的恢复
- 不需要关闭系统和重装数据库副本



(b)





数据库镜像

- 没有出现介质故障时
 - 可用于并发操作
 - 主数据和镜像可以响应不同数据部分的读请求
 - 通常写请求只由主数据响应，并复制到镜像中
- 频繁地复制数据自然会降低系统运行效率
 - 在实际应用中用户往往只选择对**关键数据**和**日志文件**镜像
 - 不是对整个数据库进行镜像





祝各位同学
身体健康，
学业有成！

