

事务概念复习

11.1 事务 (transaction)

1、定义

构成一个独立逻辑工作单位的数据库操作集。

2、构成方式

显式、隐式

3、事务的ACID性质

1) 原子性 (Atomicity)

2) 一致性 (Consistency)

① 定义

事务的执行必须是将DB从一个正确（一致）状态转换到另一个正确（一致）状态。

事务概念复习

2) 一致性 (consistency)

② 目标

保证DB数据正确性（防止丢失更新、读脏、读不可重复）。

③ 技术

并发控制。

④ 实现

用户定义事务（保证相关操作在一个事务中）；

DBMS负责维护事务执行导致数据库状态变化过程中的一致性。

事务概念复习

3) 隔离性 (isolation)

① 定义

一个事务中对**DB**的操作及使用的数据与其它并发事务无关，并发执行的事务间不能互相干扰。

② 目标

避免链式干扰。

③ 技术

并发控制。

④ 实现

DBMS依据应用程序设定的事务隔离级别自动实现。

插入

更新

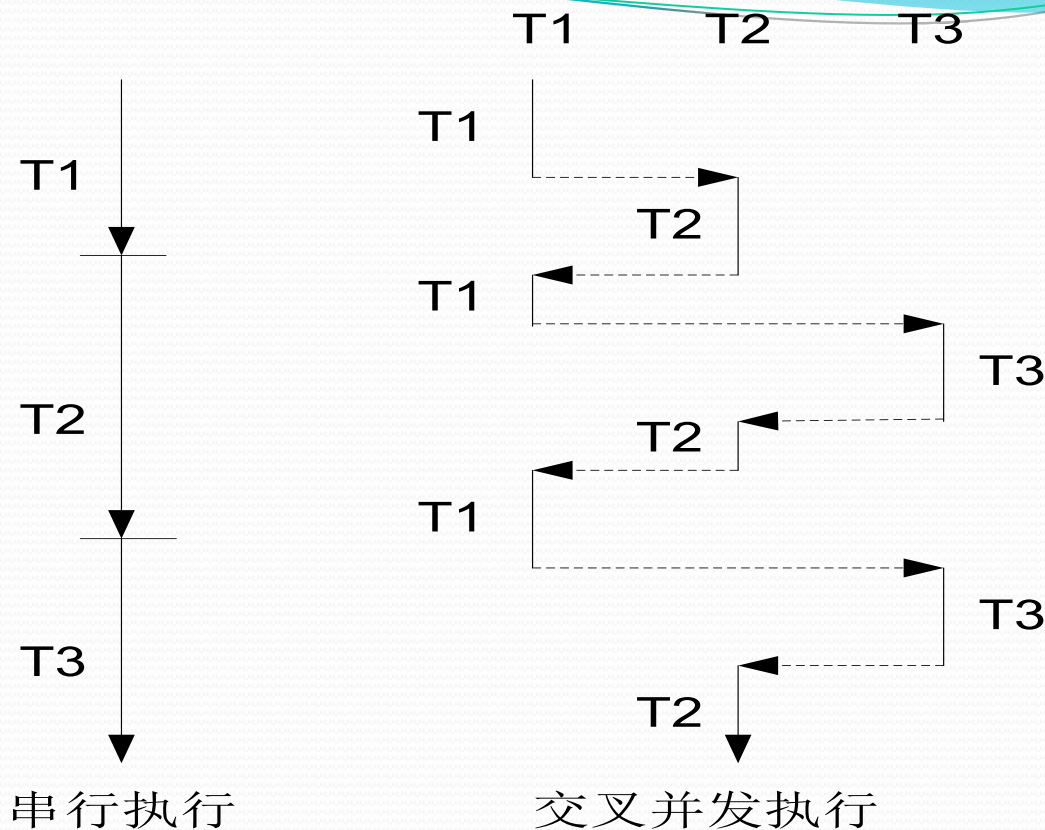
插入

4) 持久性 (Durability)



11.2 并发控制

- 事务有ACID特性，在恢复章节谈到用日志、影子页保证原子性，本章重点阐述如何解决隔离性与一致性问题。
- 一致性分两类
 - ✓ 数据库一致性（Database Consistency）
 - ✓ 事务一致性（Transaction Consistency）
- 采用封锁、协议保证隔离性
 - ✓ 悲观（Pessimistic）
 - ✓ 乐观（Optimistic）



1、问题的提出

问题1) 丢失更新 (lost update)

——两个以上事务从DB中读入同一数据并修改之，其中一事务的提交结果破坏了另一事务的提交结果，导致该事务对DB的修改被丢失。

例:

时间	T_A	DB 中 X 值	T_B
T1	$R(X)=100$	100	\circ \circ \circ
T2	\circ \circ \circ		$R(X)=100$
T3	$X:=X-1$ $W(X)=99$ COMMIT	99	\circ \circ \circ
T4		99	$X:=X-1$ $W(X)=99$ COMMIT

注:

$R(X)=100$ 表示

从 DB 中读入 X 值 100,

$W(X)=99$ 表示更新数据项 X 的值为 99。

按ti执行:

- **TA**在**t1**处从**DB**中读入**X**值（为**100**），**TB**在**t2**处读入值（**100**）
- **TA**在**t3**执行**X-1**并写回**DB**，**DB**中**X**值为**99**
- **TB**在**t4**执行**X-1**并写回**DB**，**DB**中**X**值为**99**。
- **TB**对**X**的修改覆盖了**TA**对**X**修改，使**TA**之修改丢失。
（若为飞机订票，则**TA**、**TB**实卖**2**张票，但系统中只表现卖了一张票）

调度的文本表达方式:

时间	T_A	DB 中 X 值	T_B
T1	$R(X)=100$	100	◦ ◦ ◦
T2	◦ ◦ ◦		$R(X)=100$
T3	$X := X-1$ $W(X)=99$ COMMIT	99	◦ ◦ ◦
T4		99	$X := X-1$ $W(X)=99$ COMMIT

注:

 $R(X)=100$ 表示

从 DB 中读入 X 值 100,

 $W(X)=99$ 表示更新数据项 X 的值为 99。

$$Sc1 = R_A(X) R_B(X) W_A(X) W_B(X)$$

问题2) 读不可重复 (non-repeatable read)

——同一事务重复读同一数据，但获得结果不同。

① 一事务读取后，另一事务对之进行了修改

例：

时间	T_A	DB 中值	T_B
T1	R(A)=50 R(B)=100 C: =A+B	50 (A) 100 (B)	
T2		50 (A) 200 (B)	R(B)=100 W(B)=B*2 COMMIT
t3	R(A)=50 R(B)=200 D: =A+B	50 (A) 200 (B)	

按ti执行：

- T_A 第一次在 t_1 处读 B 为 100;
- T_B 在 t_2 处修改并写回 DB, B 为 200;
- T_A 在 t_3 处读 B (为校对用), B 为 200;
- T_A 两次读 B , (第二次为校对用), 读结果不同 (一次 100, 另一次为 200)。

② 一事务读取数据后, 另一并发事务删去了其中部分数据 (少了)。

(**幻行现象: Phantom row**)

③ 一事务读取数据后, 另一事务插入了一些新数据 (多了)。

(**幻行现象**)

调度的文本表达方式:

时间	T_A	DB 中值	T_B
T1	R(A)=50 R(B)=100 C: =A+B	50 (A) 100 (B)	
T2		50 (A) 200 (B)	R(B)=100 W(B)=B*2 COMMIT
t3	R(A)=50 R(B)=200 D: =A+B	50 (A) 200 (B)	

$$Sc2 = R_A(A)R_A(B) \text{ } R_B(B) \text{ } W_B(B) \text{ } R_A(A)R_A(B)$$

问题3) 读“脏”数据 (read dirty)

——读未提交的随后又被撤消(Rollback)的数据。

例：

时间	T_A	DB 中 x 值	T_B
t_1	$R(X)=100$ $W(X)=X*2$	100 200	
t_2		200	$R(X)=200$ (读 x 为 200)
t_3	Rollback ($W(X)=100$)	100	

SC3= $R_A(X)$ $W_A(X)$ $R_B(X)$ $W_A(X)$

接上例：SC3= $R_A(X) \ W_A(X) \ R_B(X) \ W_A(X)$

按ti执行：

- T_A 在 t_1 读 x (100)并修改为200后写回磁盘DB中；
- T_B 在 t_2 读 x 为200；
- T_A 在 t_3 处撤消对 x 修改， x 恢复为100；
- T_B 此时读出的 x (200) 数据与DB实际值(100)不一致，即TB读到的不正确的“脏”数据。

2. 上述几类问题的原因

多事务并发操作数据库，破坏了事务的隔离性，导致了数据不一致。

3. 方法

并发控制(正确的并发操作调度策略)。

11.2.1 概述

1、什么是封锁？

并发控制的一种技术。

并发控制的多种方法：

- ① 锁（**Locking**）——商用主要方法
- ② 乐观（**Optimistic**）
- ③ 时间戳（**timestamping**）

2、封锁规则

- ① 将要存取的数据须先申请加锁；
- ② 已被加锁的数据不能再加不相容锁；
- ③ 一旦退出使用应“立即”释放锁；
- ④ 未被加锁的数据不可对之解锁。

11.2.2 申请时机

1、事务

- 无死锁；
- 锁开销少；
- 并发性低。

2、一条SQL语句

- 并发性高；
- 锁开销大；
- 死锁；
- 申请频繁。

11.2.3 申请方式

1、显式

应事务的要求直接加到数据对象上

2、隐式

该数据对象没有独立加锁，由于数据对象的多粒度层次结构中的上级结点加了锁，使该数据对象隐含的加了相同类型的锁。



11.2.4 封锁类型

1、排它锁（X锁：exclusive lock）

——若事务 T_i 持有数据 D_i 的X锁，则 T_i 可读、写 D_i ，其它任何事务不能再对 D_i 加任何锁，直至 T_i 释放该X锁。

X锁用于写保护，防止丢失更新。

相容矩阵：

	-	X	-：无锁
-	Y	Y	X：排它锁
X	Y	N	Y：相容 N：不相容

2、共享锁（S锁：share lock）

——若事务Ti持有数据Di的S锁，则其它事务仍可对Di加S锁，但不可加X锁，直到Ti释放该S锁。

一旦施加S锁，读可共享，但其它事务不可改。

S锁用于读操作保护。

相容矩阵：

	-	S	S：共享锁
-	Y	Y	Y：相容
S	Y	Y	

3、封锁类型的相容矩阵

$T_1 \backslash T_2$	X	S	—	Y 相容的请求 N 不相容的请求
X	N	N	Y	
S	N	Y	Y	
—	Y	Y	Y	

对应问题1：加锁解决丢失更新

时间	T _A	X 值	T _B	说明
t1	X Lock X R(X)=100	100		T _A 对 X 加锁成功 后读 X
t2			<u>X Lock X</u> 等待	T _B 对 X 加 X 锁未 成功则等待
t3	W(X)=X-1 COMMIT UnLock X	99	等待	T _A 修改, X 结果写 回 DB 释放 X 锁
t4			X Lock X R(X)=99 W(X)=X-1 COMMIT UnLock X	T _B 获得 X 的 X 锁 读 X 得 99 (已更 新后结果) 将修改 后 X(98) 写回 DB

对应问题3：加锁解决读脏

时间	T _A	X 值	T _B	说明
t1	X Lock X R(X)=100 W(X)=X*2	100 200		T _A 先获得 X 锁
t2			S Lock X 等待	T _B 申请 S 锁 T _A 未释放, T _B 等待
t3	Rollback (W(X)=100) UnLock X	100		T _A 因故撤消 X 值恢复为 100
t4			S Lock X R(X)=100	T _B 获得 S 锁 T _B 读到值与 DB 中值一致 防止了读脏

对应问题2：加锁解决不可重复读

时间	T _A	DB 中 A、B 值	T _B	说明
t1	S Lock A R(A)=50 S Lock B R(B)=100 C = A+B	A: 50 B: 100 A: 50		T _A 对 A、B 加 S 锁 T _B 不能再对之加 X 锁
t2			X Lock B 等待	T _B 对 B 加锁不成功 T _B 等待
t3	R(A)=50 R(B)=100 COMMIT Unlock A Unlock B	A: 50 B: 100		T _B 等待 T _B 重读 A、B 计算、结果相同
t4		A: 50 B: 200	X Lock B R(B)=100 W(B): =B*2 写回 B=200	T _B 获得 B 的 X 锁

11.3 三级封锁协议

1) 1级封锁协议

① 策略

事务 T_i 在修改数据 D_i 之前须先对 D_i 加X锁，直到事务 T_i 结束（commit/rollback）才释放。

② 功能

防止丢失修改；

保证 T_i 可恢复（若意外终止，则rollback后才可释放）。

③ 问题

不能防止读不可重复和读“脏”数据。

（1级协议仅对修改操作，若读则不加锁）

2) 2级封锁协议

① 策略

在1级封锁协议加上事务 T_i 在读取 D_i 之前领先对 D_i 加S锁，读完后即可释放该S锁。

② 功能

- 防止丢失修改;
- 防止读脏。

③ 问题

不能防止读不可重复（读完即释放，重读可能其它事务对之修改）。

3) 3级封锁协议

① 策略

在1级封锁协议上加上Ti读Di前须先对Di加S锁，直至Ti结束后才释放该S锁。

② 功能

- 防止丢失修改；
- 防止读“脏”；
- 防止读不可重复。

4) 总结

三级封锁协议异同

	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 ”	可重 读
1 级 协议		✓			✓		
2 级 协议		✓	✓		✓	✓	
3 级 协议		✓		✓	✓	✓	✓

11.4 活锁和死锁

11.4.1 活锁 (live lock)

1、含义

——事务因故永远处于等待状态。可能因为相互之间的竞争不断循环“重启-竞争”的过程。

“饥饿” ???

2、方法

FCFS——先来先服务，或者错峰避免频繁冲突。

11.4.2 死锁 (dead lock)

1、含义

——两个或两个以上事务均处于等待状态，每个事务都在等待其中另一个事务封锁的数据，导致任何事务都不能继续执行的现象称为死锁。

时间	T_A	T_B
t1	X Lock A	
t2		X Lock B
t3	X Lock B 等待	
t4		X Lock A 等待
t5	等待	等待

2、产生条件

- ① 互斥（排它性控制）；
- ② 不可剥夺（释放前，其它事务不能剥夺）；
- ③ 部分分配（每次申请一部分，申请新的时，仍占用已获得者）；
- ④ 环路（循环链中，每事务获得数据同时又被另一事务请求）。

3、死锁处理

1) 预防

——防止产生条件之一发生。

① 一次封锁法

——每个事务事先一次获得其需数据的全部锁。

如： T_A 获得所有数据A、B锁， T_A 连续执行， T_B 等待； T_A 执行完后释放A、B锁， T_B 继续执行，不会发生死锁。

特征： 简单； 无死锁； 粒度大，并发性低；

难以确定封锁对象（DB常变化，只好扩大封锁范围）。

② 顺序封锁法

——事务按预先确定的数据封锁顺序实行封锁。

上例中：

设封锁顺序： $A \rightarrow B$ ；

T1、T2均按此顺序申请锁；

若T1先获得A、B锁；

T2则先申请A锁，等待；

T1释放A、B锁；

T2获得锁运行。

特征：无死锁； 顺序难以维护； 封锁对象难以确定（运行时确定封锁对象）。



死锁的预防_时间戳优先级法

根据时间戳分配优先级:

→ 较老时间戳 = 高优先级 (例如 $T_1 > T_2$)

Wait-Die ("Old Waits for Young")

→ 如果请求锁事务比持有锁的事务具有更高的优先级 (更早), 则请求事务等待其完成。

交错时等待

→ 否则, 请求事务撤销。

Wound-Wait ("Young Waits for Old")

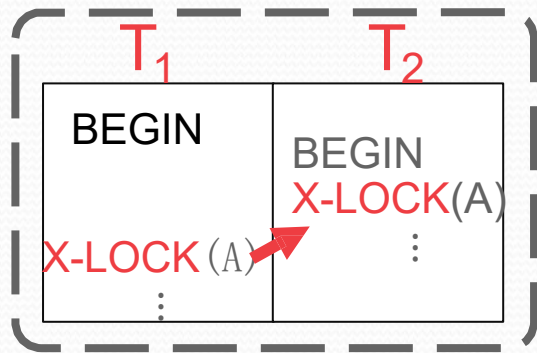
→ 如果请求的事务比持有锁的事务具有更高的优先级, 则撤销持有锁的事务, 释放锁。

交错时夭折对方

→ 否则, 请求事务等待



死锁的预防_时间戳优先级法



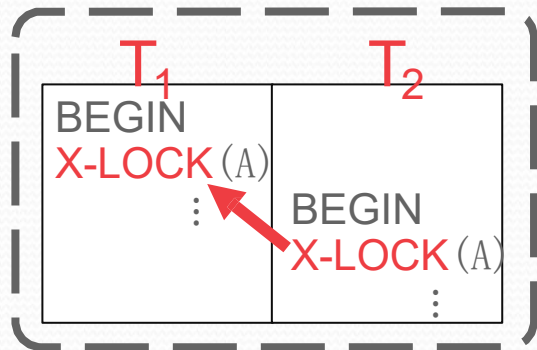
Wait-Die

T_1 waits

Wound-Wait

T_2 aborts

Old wait for young



Wait-Die

T_2 aborts

Wound-Wait

T_2 waits

Young wait for Old



2) 死锁的诊断与解除

2.1) 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. 2) 等待图法

- ① 构造一事务等待图;
- ② 周期性检测该等待图;
- ③ 判断存在环路否;
- ④ 存在, 则撤消某一事务。



(2.2) 等待图法

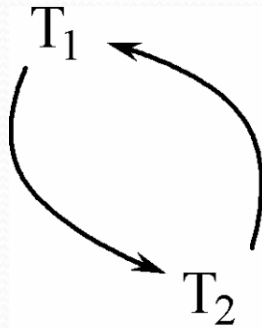
用事务等待图动态反映所有事务的等待情况

事务等待图：是一个有向图 $G=(T, U)$

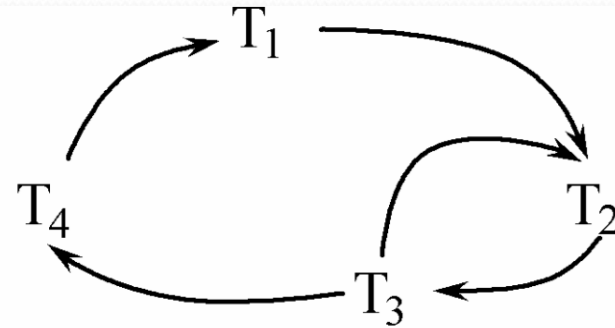
- T 为结点的集合，每个结点表示正运行的事务
- U 为边的集合，每条边表示事务等待的情况
- 若 T_1 等待 T_2 ，则 T_1, T_2 之间划一条有向边，从 T_1 指向 T_2



(2.2) 等待图法 (续)



(a)



(b)

事务等待图

- 图(a)中，事务 T_1 等待 T_2 ， T_2 等待 T_1 ，产生了死锁
- 图(b)中，事务 T_1 等待 T_2 ， T_2 等待 T_3 ， T_3 等待 T_4 ， T_4 又等待 T_1 ，产生了死锁
- 图(b)中，事务 T_3 可能还等待 T_2 ，在大回路中又有小的回路



解除死锁

如何选择合适的业务撤销取决于多种因素...

- 事务年龄（最小的时间戳）
 - 事务进度（执行最少/最多查询）
 - 所锁定的数据库对象数量
 - 需回滚事务的数量
- 还需考虑事务重新启动的次数，以防事务饿死。

11.5 并发操作调度的可串行性

11.5.1 正确性标准（可串行化调度）

1) 单个事务

——若非并发的执行，每个事务都能保证数据库的正确性。
（上述问题，都是因事务并发执行产生）

2) 多个事务

——多个事务以任意串行方式执行都能保证数据库的正确性。

给定三个事务： T_1 ， T_2 ， T_3 。

$T_1 \rightarrow T_2 \rightarrow T_3$
 $T_1 \rightarrow T_3 \rightarrow T_2$
 $T_2 \rightarrow T_1 \rightarrow T_3$
 $T_2 \rightarrow T_3 \rightarrow T_1$
 $T_3 \rightarrow T_1 \rightarrow T_2$
 $T_3 \rightarrow T_2 \rightarrow T_1$

显然，任何一事务并发执行时禁止其它事务执行，总能保证数据库正确性，但不利于数据共享。

3) 可串行化调度 (serializability)

——当且仅当多个事务并发执行的结果与这些事务按某一顺序串行执行的结果相同时，则该并发执行是可串行化的。

(可串行化调度是并发事务正确性的唯一准则)

例：有两个事务 T_A , T_B ($A=10, B=2, C=0$) 包含如下操作序列

T_A : 读B; $A:=B+1$; 写回A;

T_B : 读C; 读A; $B:=A+1$; 写回B;

则至少可能有四种不同的调度方式，

然而，正确的结果只有两种： $T_A T_B$ 的结果，或者 $T_B T_A$ 的结果。

① 串行调度1

时间	T_A	DB 中值	T_B
t_1	$R(B)=2$ $A=B+1$ $W(A)=3$	2 (B 初值) 3(A)	
t_2		3(A) 4(B)	$R(C)=0$ $R(A)=3$ $B=A+1$ $W(B)=4$

先执行 T_A ，再执行 T_B —— $R_A(B)W_A(A)R_B(C)R_B(A)W_B(B)$;
 结果: $A=3$, $B=4$ 。

② 串行调度2

时间	T_A	DB 中值	T_B
t_1		10 (A 初值) 11(B)	$R(C)=0$ $R(A)=10$ $B=A+1$ $W(B)=11$
t_2	$R(B)=11$ $A=B+1$ $W(A)=12$	12(A) 11(B)	

先执行TB，再执行TA——

$R_B(C)R_B(A)W_B(B)R_A(B)W_A(A)$;

执行结果：A=12，B=11。

时间	T_A	DB 中值	T_B
t_1	$R(B)=2$	2(B 初值)	
t_2		2(B) 10(A 初值)	$R(C)=0$ $R(A)=10$
t_3	$A=B+1$ $W(A)=3$	2(B) 3(A)	
t_4		11(B) 3(A)	$B=A+1$ $W(B)=11$

两事务 T_A 、 T_B 按 t_i 并发执行—— $R_A(B)$ $R_B(C)$ $R_B(A)$ $W_A(A)$ $W_B(B)$, 结果为 **$A=3$** , **$B=11$** 。

按事务并发可串行化的正确性准则, 本结果错误, 其结果与 T_A 、 T_B 两个串行执行的任何结果 (**$A=3$** , **$B=4$** ; **$A=12$** , **$B=11$**) 均不同。

④ 交错执行(可串行化)调度

时间	T_A	DB 中值	T_B
t_1	$R(B)=2$	2(B 初值)	
t_2			$R(C)=0$
t_3			等待
t_4	$A=B+1$ $W(A)=3$	2(B) 3(A)	等待
t_5		3(A) 4(B)	$R(A)=3$ $B=A+1$ $W(B)=4$

按 t_i 交错执行—— $R_A(B) R_B(C) W_A(A) R_B(A) W_B(B)$ ；执行结果： **$A=3$** ， **$B=4$** 。

该结果正确，因为与串行化调度1结果相同，该调度是可串行化调度。

11.5.2 冲突可串行化调度

——可串行化调度的充分而非必要条件。



■ 冲突操作

- ① $R_i(x)$ 与 $W_j(x)$ ② $W_i(x)$ 与 $W_j(x)$

■ 操作顺序的交换（可交换、不可交换）

——不同事务的冲突操作和同一事务的两个操作均是不可交换的。否则可能使操作序列的结果不等价。

■ 在可交换的前提下，若干事务的操作交换顺序的结果是一个串行调度，则称这些事务是冲突可串行化的。



例: $Sc1 = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

经过冲突等价的操作交换, 得到如下调度序列

$Sc2 = r_1(A)w_1(A) r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

显然, $Sc2$ 等价于先执行 $T1$ 再执行 $T2$ 的串行调度, 因此, $Sc1$ 是一个冲突可串行化的调度。

例: 有三个事务 $T1=W1(Y)W1(X)$, $T2=W2(Y)W2(X)$, $T3=W3(X)$, 两种调度方案:

$L1 = W_1(Y)W_1(X)W_2(Y)W_2(X)W_3(X) W_3(Y)$

$L2 = W_1(Y)W_2(Y)W_2(X)W_1(X)W_3(X) W_3(Y)$

冲突可串行化是
必要条件么?

能作为调度正确
性实施标准么?

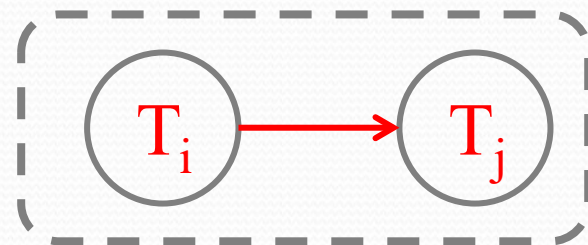
$L1$ 是一个串行调度, $L2$ 不能实现冲突可串行化, 但是 $L2$ 是可串行化的, 因为其结果等价于 $L1$ 的结果。

问题思考

- 冲突可串行化：在完全验证可串行化难以实施的情况下寻找接近答案的解决方案
- 还有没有更好的等价方案？
视图可串行化

依赖图

- 每个节点代表一个事务



- 从 T_i 节点到 T_j 节点的边表示：

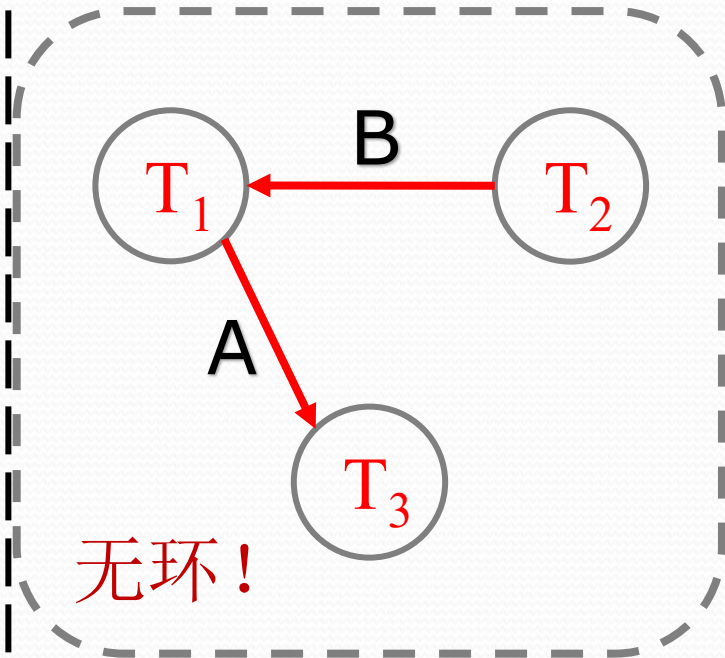
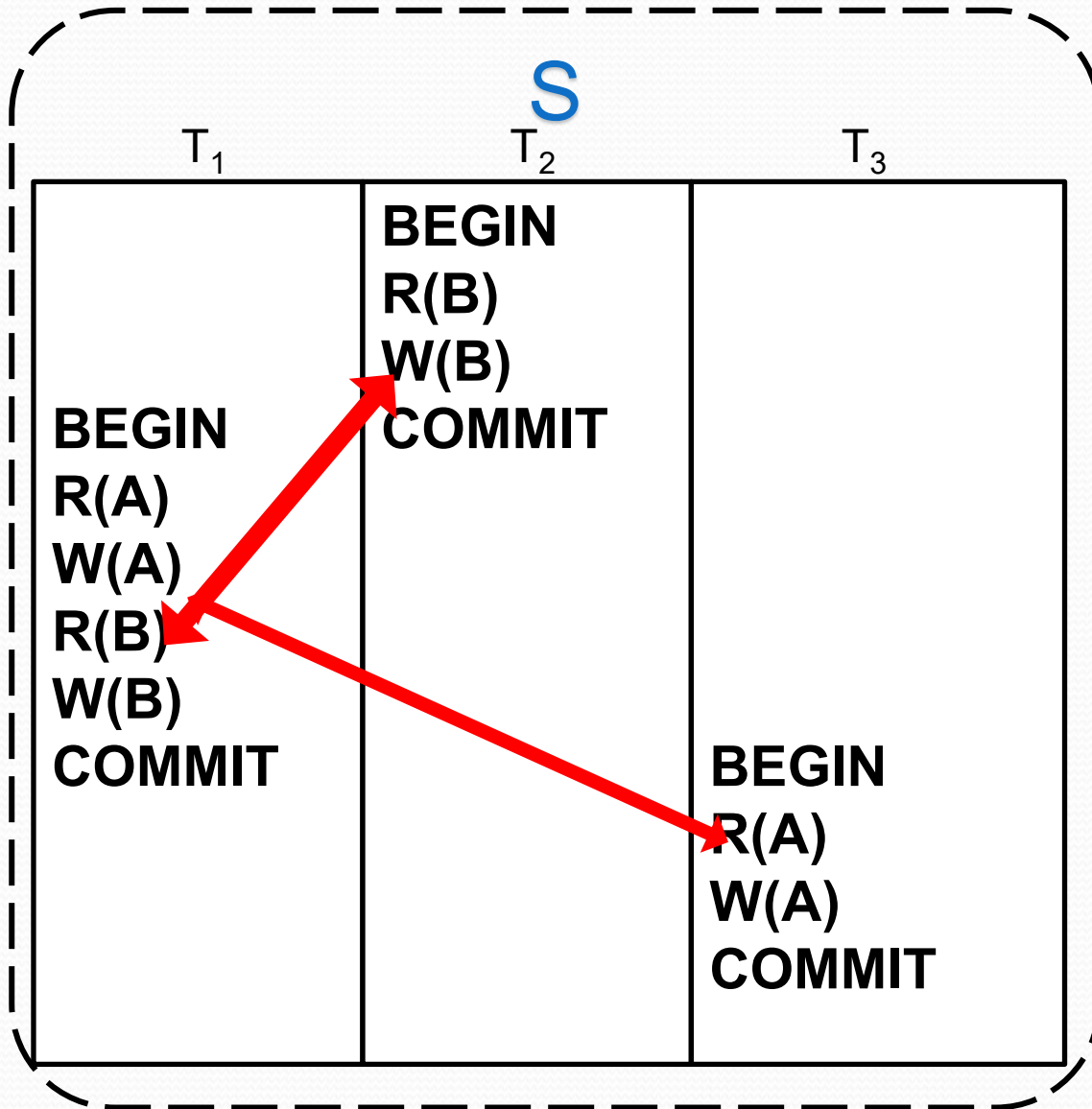
T_i 中的操作 O_i 发生在 T_j 中的操作 O_j 之前，且 O_i 和 O_j 是冲突的。

时序

如果最后得到一个有向无环图，则结果是正确的！

时序

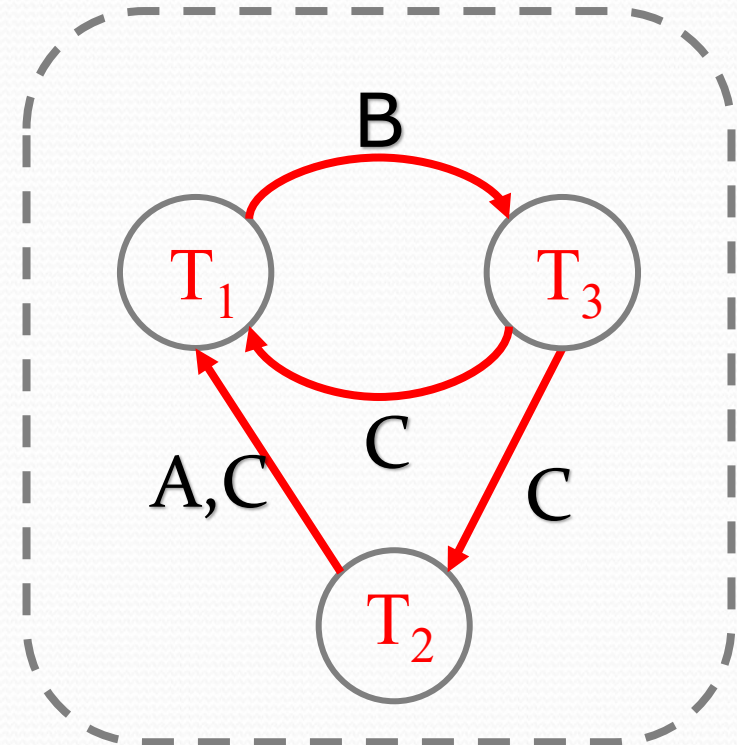
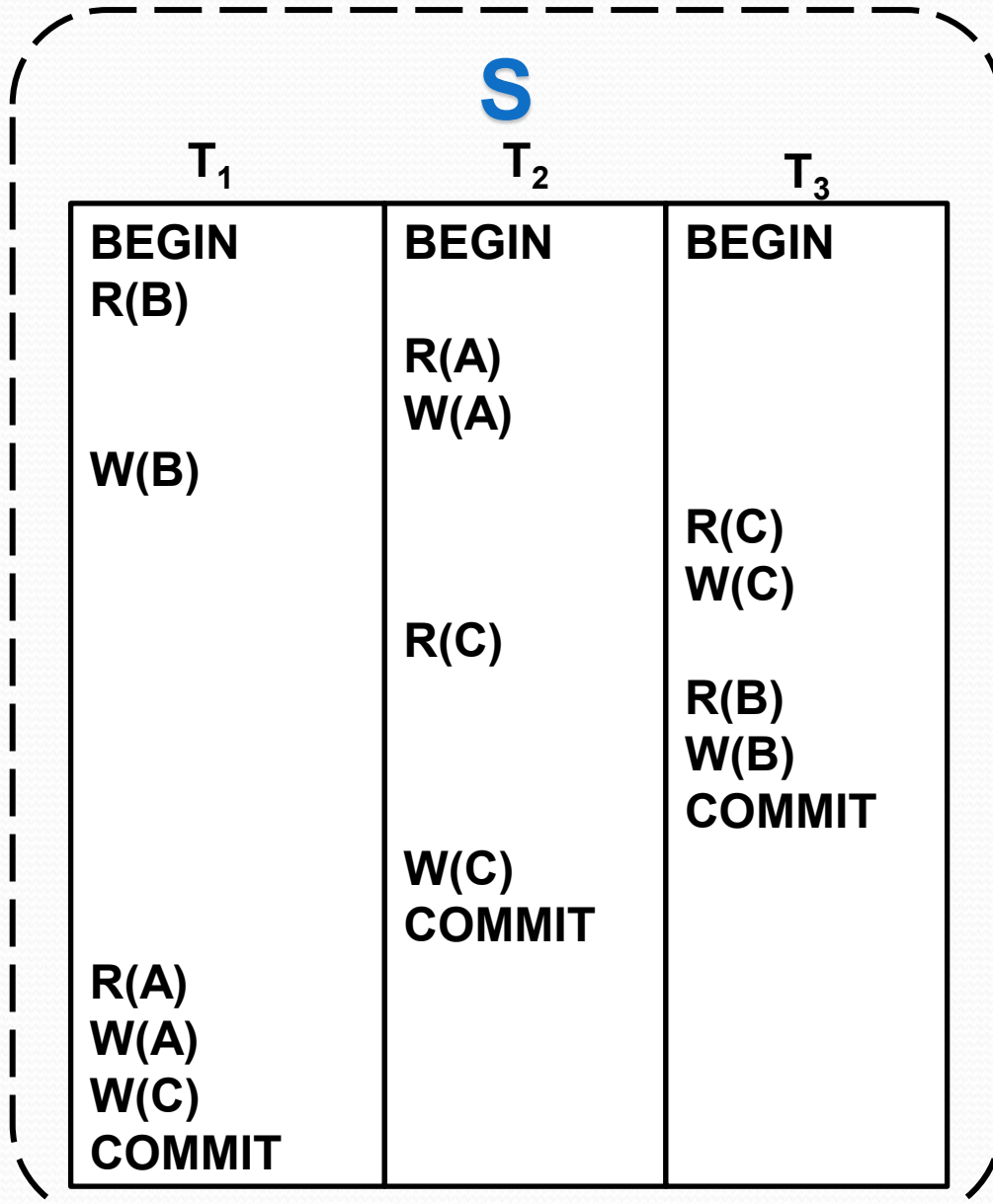
依赖图



串行调度的依赖图一定是有向无环图。

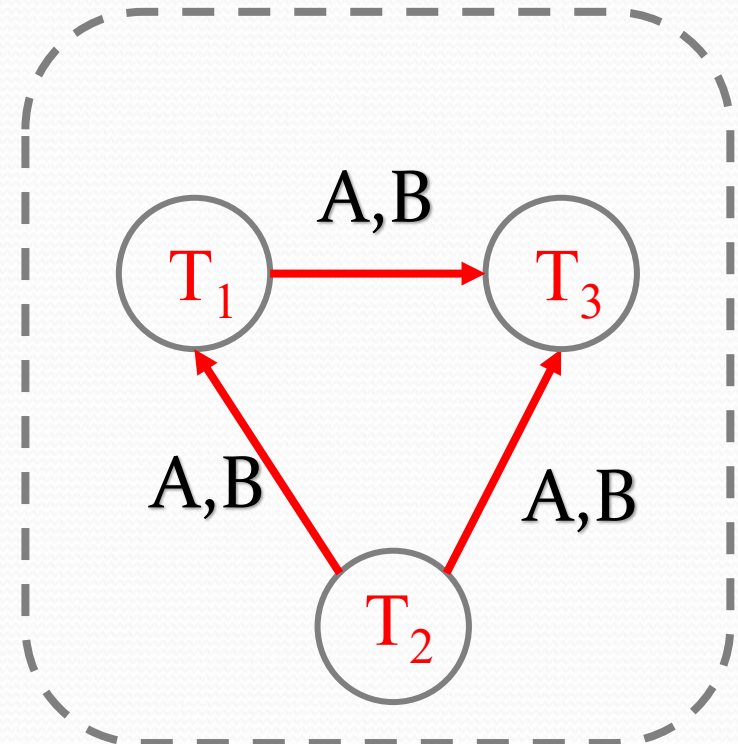


依赖图——练习1



依赖图——练习2

S		
T ₁	T ₂	T ₃
BEGIN R(A) A=A-10 W(A) R(B) B=B+10 W(B) COMMIT	BEGIN R(A) R(B) R(C) C=C+A-B W(C) COMMIT	BEGIN R(D) D=D+20 W(D) A=D-50 W(A) R(B) B=B-10 W(B) COMMIT





视图可串行化

视图等效：满足以下3个条件的调度视图等效：

- 如果事务 T_1 在其中一个调度内读取了数据A的初始值，那么其他调度里读取数据A初始值。初始值等效
- 如果事务 T_1 在一个调度内读取数据A的值是事务 T_2 写入的，那么在其他调度里读取的也是 T_2 写入的。初始值等效
- 如果事务 T_1 在一个调度内为数据A写入了最终值，那么在其他调度内也由 T_1 为A写最终值。影响等效

基于视图的可串行化：和串行调度**视图等效**的调度是基于视图的可串行化调度。



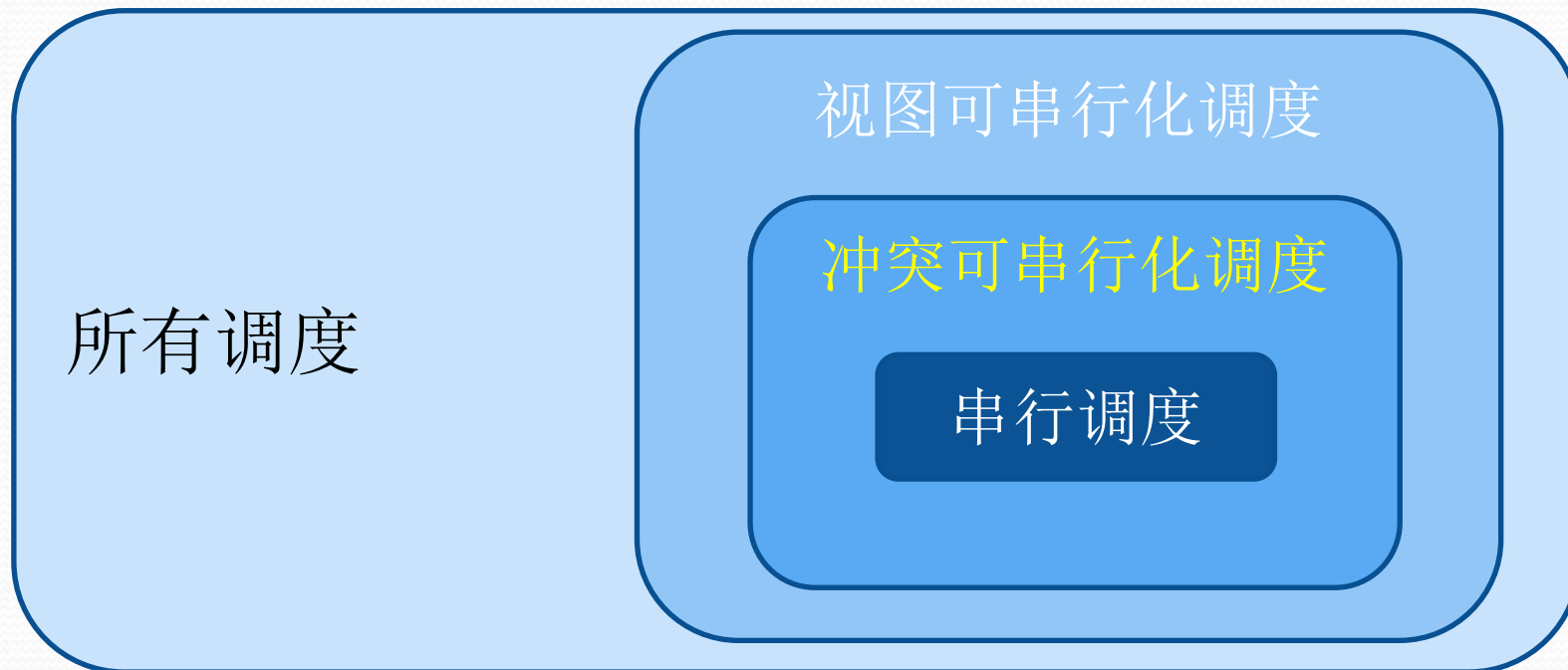
视图可串行化

视图可串行化是一个比冲突可串行化更宽松的条件，满足冲突可串行化的调度一定也满足视图（基于观察）可串行化。

遗憾的是，视图可串行化还没有行之有效的分析方法，仅依靠定义去判别依然是无法应用于实际工作的。



总结



实际应用中，系统应当尽力支持冲突可串行化的调度执行，因为冲突可串行化可以有效保障其实施，而基于视图的可串行化仍没有有效的分析方法。

11.6 两段锁（2PL: two-phase locking）协议

11.6.1 概述

1) 封锁协议的概念

——申请、持有和释放锁的规则。

2) 目的

——实现正确的并发操作调度。

3) 锁协议的类别

- ① 支持一致性维护的三级封锁协议；
- ② 支持并发调度可串行化的两段锁协议；
- ③ 避免死锁协议。

11.6.2 两阶段锁

1) 含义

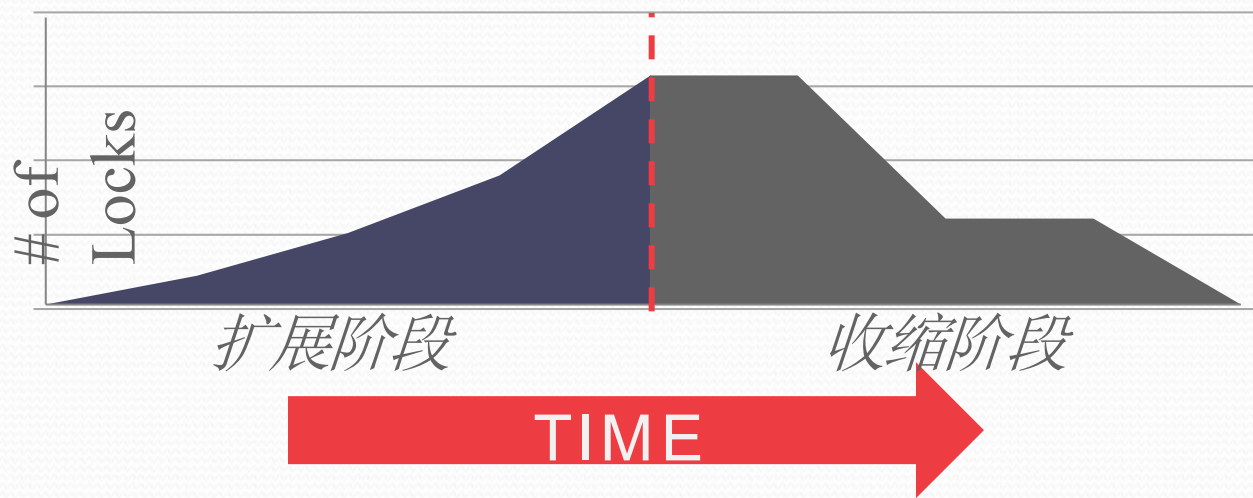
事务分为两个阶段，第一阶段称为**扩展阶段**（获得锁）；第二阶段称为**收缩阶段**（释放锁）。



两阶段锁协议

在扩展阶段之后，事务不允许获得/更新锁。

事务生命期



例：T1封锁序列：

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

正确的遵守2PL协议，所有获得锁均在释放锁之前。

例：T2封锁序列：

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

不正确（未遵守2PL协议）：不是所有申请锁均在释放之前。

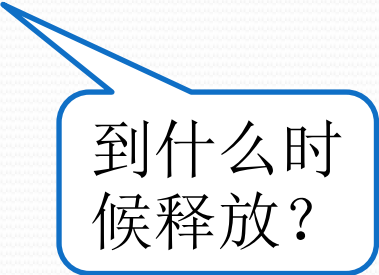
2) 策略

- ① 在对任何数据读、写之前，须先获得该数据锁（且）；
- ② 在释放一个封锁之后，该事务不能再申请任何其它锁。

3) 目标

实现并发操作调度的可串行化。

（释放一个锁之后又继续去获得另一个锁的事务仍然可能产生错误结果）



到什么时候释放？

4) 定理

若所有事务都**严格**遵守2PL协议，则对这些事务的所有并发调度策略都是可串行化的。

证明步骤：

- ① 按Lock、Unlock操作中**冲突等待关系的拓扑结构**建立有向图G；
- ② 假设不是可串行化调度；
- ③ 在锁机制的背景下，G中必存在冲突环路（**虫洞定理**）：
 $T_{i1} \rightarrow T_{i2} \rightarrow \dots \rightarrow T_{jp} \rightarrow T_{i1}$ ；其中某个冲突事务获得锁的前提是前面的冲突事务释放锁。
- ④ T_{i1} 解锁后又有 T_{i1} 加锁；
- ⑤ “④”与 T_{i1} 的两阶段事务假设矛盾。

证毕

5) 说明

2PL协议是可串行化的充分条件，不是必要条件。
遵守两阶段锁协议的事务可能发生死锁。



两阶段锁协议的问题

2PL足够保证冲突可串行化

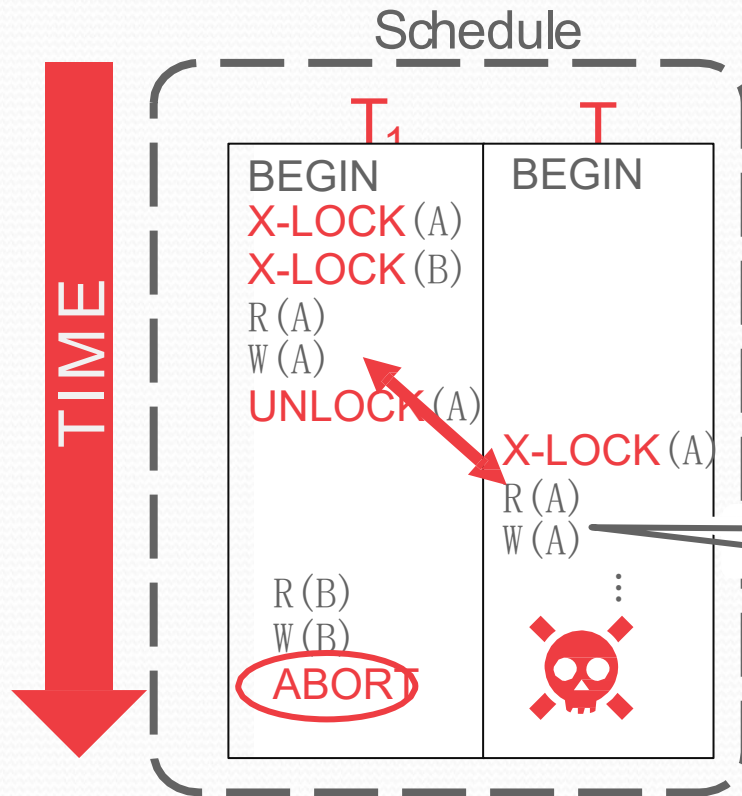
→ 所产生的调度图是无环

但存在级联撤销 (cascading abort)



2PL 级联撤销

20



根据2PL，该调度是可行的，
但当事务 T_1 撤销时，数据库
须撤销 T_2
→ T_1 的信息不能泄露给外界

This is all wasted work!

例①：2PL可串行调度

时间	T _A	DB 值	T _B	
t1	S Lock B B=2 Y=B X Lock A	B: 2		T _A 未释放 A 的 X 锁 T _B 等待 T _A 释放 B、A 锁
t2			S Lock A 等待	
t3	A: =Y+1 写回 A=3 UNLock B UNLock A	B: 2 A: 3		
t4		B: 2 A: 3	S Lock A A=3 Y=A	T _B 获得锁
t5		B: 4 A: 3	X Lock B B=Y+1 写回 B=4 UNLockB UNLockA	

T_A、T_B各自的所有申请获得锁均在释放之前，符合2PL协议的可串行调度。

例② 不符合2PL可串行化调度

时间	T _A	DB 值	T _B
t1	S Lock B B=2 Y: =B UNLock B X Lock A	B=2	
t2			<div>S Lock A</div> 等待
t3	A: =Y+1 写回 A=3 UNLock A		等待
t4		A=3 B=4	S Lock A A=3 Y: =A UN Lock A X Lock B B: =Y+1 写回 B=4 UNLock B

显然不符合2PL协议
结果 $A=3$, $B=4$
可串行化调度。

6) 2PL类型

问题：可能存在“脏读”。

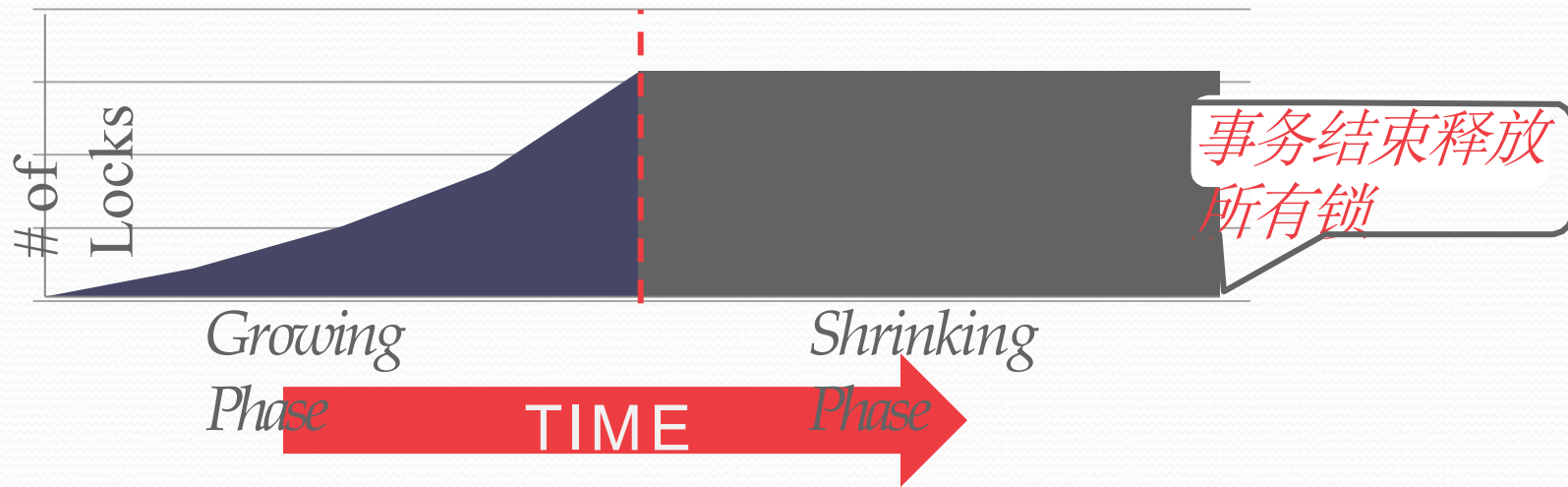
→ 解决方法：强严格2PL (Strong Strict 2PL, SS2PL)



强严格2PL

22

扩展阶段后，事务不允许获得/更新锁。
只允许可串行化调度, 对于某些应用来说，它比所需要的要更强。





强严格2PL

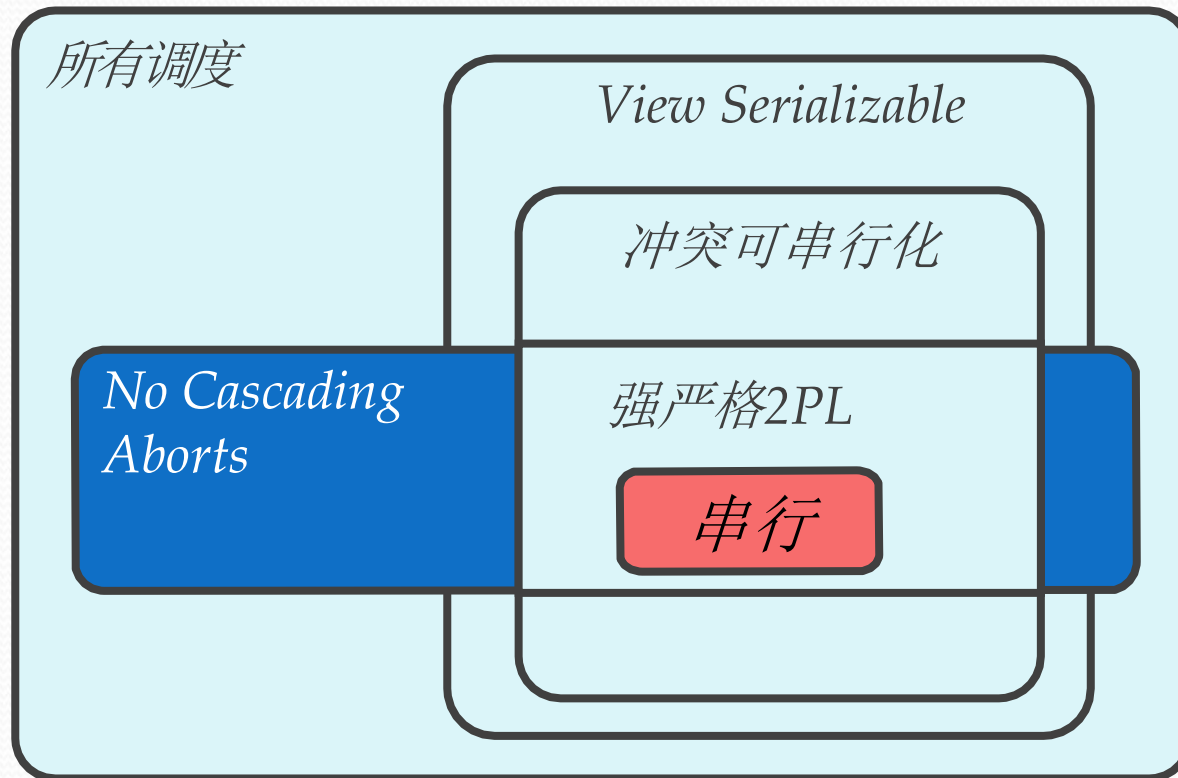
调度是严格的：如果在一**事务结束前**，该事务所**写的值**不会被其它事务读或写。

优点：

- 不会引发级联撤销。
- 撤销事务只需恢复修改元组的原始值。



调度



11.7 封锁粒度 (granularity)

——被封锁数据的范围。

1、逻辑单元

整个DB、整个关系、整个索引、元组、索引项、属性值集、属性值。

2、物理单元

块、数据页、索引页。

3、评价

1) 粒度大：被封锁对象少，并发性差，开销小。

2) 粒度小：被封锁对象多，并发性高，开销大。

4、一般策略

- 1) 需常存取多个关系的大量元组时宜采用**DB级**粒度;
- 2) 需常存取单个关系大量元组时宜采用**关系级**;
- 3) 需常存取单个关系少量元组时宜采用**元组级**;
- 4) 一般不采用**属性级**;
- 5) **物理单元一般不宜采用**。

5、一般规则

- 1) 锁住了大范围, 则不再申请锁住其中部分;
- 2) 反之亦然。

多粒度锁

- 封锁协议

对结点的加锁意味着后裔结点也被加以同样类型的锁。

- 封锁方式 （回顾）

- 1) 显式——直接施加在结点本身

- 2) 隐式——由于上级结点的封锁导致的本节点隐含着施加了相同类型的锁。

多粒度锁申请的授予条件：

- 1) 检查数据对象上是否有显式封锁与之冲突；
- 2) 检查上级结点上是否有封锁与本结点冲突；
- 3) 检查下级结点上是否有封锁与本结点冲突。

校验已有的
隐式封锁冲突

校验欲施加的
隐式封锁的冲突，**开销大**

6、意向锁

如果对某个结点加意向锁，则表示该结点的某个子孙结点正在或拟施加相应的非意向锁。对任一结点加锁时，必须**先对它的上层结点加意向锁**。——提高系统并发度，减少加锁和解锁的开销，被商用产品广泛采用。

1) IS锁

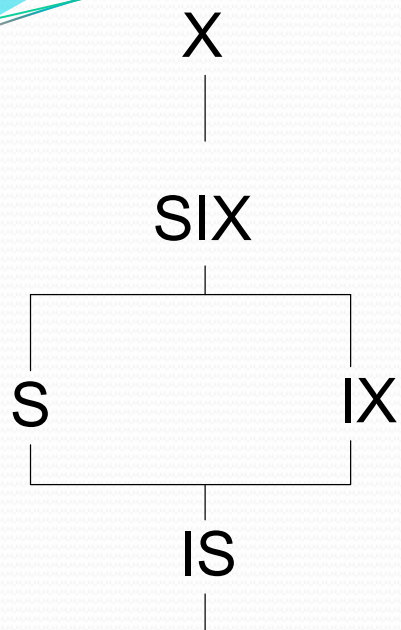
表示某个子孙结点拟加S锁

2) IX锁

表示某个子孙结点拟加X锁

3) SIX锁

表示对结点施加S锁的同时，再施加IX锁。



锁强度偏序关系

T1 \ T2	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

锁相容矩阵

*11.8 其他并发控制

时间戳具有唯一性和单调性

11.8.1 时间戳方法

- 给每个事务赋予一个时标（时间戳）——事务开始执行的时间。
- 每个数据也被赋予读和写两个时间戳。
- 按照时间戳来解决事务的冲突操作：当有冲突发生时，回滚具有较早时间戳的事务，以保证其他事务的正常执行，被回滚的事务被赋予新的时间戳并从头开始执行。

(1) 假设事务 T_i 发出 $\text{read}(Q)$ 操作：读/写冲突

- ① 若 $\text{TS}(T_i) < W\text{-TS}(Q)$ ，则 T_i 需要读入的 Q 值已被覆盖。因此， read 操作被拒绝， T_i 回滚；
- ② 若 $\text{TS}(T_i) \geq W\text{-TS}(Q)$ ，则执行 read 操作，且更新 $R\text{-TS}(Q)$ 的值为 $R\text{-TS}(Q)$ 与 $\text{TS}(T_i)$ 中的较大者。

(2) 假设事务 T_i 发出 $\text{write}(Q)$ 操作：写/读和写/写冲突

- ① 若 $\text{TS}(T_i) < R\text{-TS}(Q)$ ，则 T_i 产生的 Q 值是先前时间戳所需要的值，且系统已假定该值不会被产生。因此， write 操作被拒绝， T_i 回滚；
- ② 其他情况下执行 write 操作，并将 $W\text{-TS}(Q)$ 的值设为 $\text{TS}(T_i)$ 。

11.8.2 乐观控制法

- 乐观的认为事务执行时很少发生冲突，因此不对事务进行特殊的管制，任其自由执行，事务提交前再进行正确性检查。
- 若检查发现冲突并影响了可串行性，则拒绝提交并回滚该事务，乐观控制法又称为验证方法（**certifier**）。

11.8.3 多版本并发控制

- 版本（**version**）指数据库中数据对象的一个快照，记录数据对象某个时刻的状态。
- **write(Q)**操作创建Q的一个新版本，导致Q有一个版本序列 Q_1, Q_2, \dots, Q_m 。
- 每个版本 Q_k 拥有版本值、创建它的事务的时间戳**W-timestamp(Q_k)**、读取 Q_k 的事务的最大时间戳**R-timestamp(Q_k)**。
- **TS(T)**表示事务T的时间戳。

11.8.3 多版本并发控制

- 寻找 $TS(T)$ 之前的最大 Q_k 。
- 若 T 要读取 Q ，则获取 Q_k 。
- 若 T 要写 Q ，则：

当 $TS(T) < R\text{-timestamp}(Q_k)$ 时， T 回滚；

当 $TS(T) = W\text{-timestamp}(Q_k)$ 时，覆盖 Q_k ；

否则，创建 Q 的新版本。

T 开始后，有其它事务读取了 Q_k 版本。
一致性读？

Q_k 版本是 T 创建的。

11.8.3 多版本并发控制

- 如果一个数据对象 Q 的两个版本 Q_k 和 Q_t ，其创建时间戳（W-timestamp）都小于系统中最老的事务的时间戳，则 Q_k 和 Q_t 中W-timestamp较小的将不再被用到。

多版本并发控制

- 寻找TS(T)之前的最大 Q_k 。
- 若T要读取 Q ，则返回 Q_k 。
- 若T要写 Q ，则：

当 $TS(T) < R\text{-timestamp}(Q_k)$ 时，T回滚；

T开始后，有其它事务读取了 Q_k 版本。
一致性读？

11.8.4 MV2PL多版本控制

- 事务分为只读和更新两类。
- 除了读锁（**R锁**）和写锁（**W锁**），引进一个新的封锁类型——验证锁（**certify-lock, C锁**）。

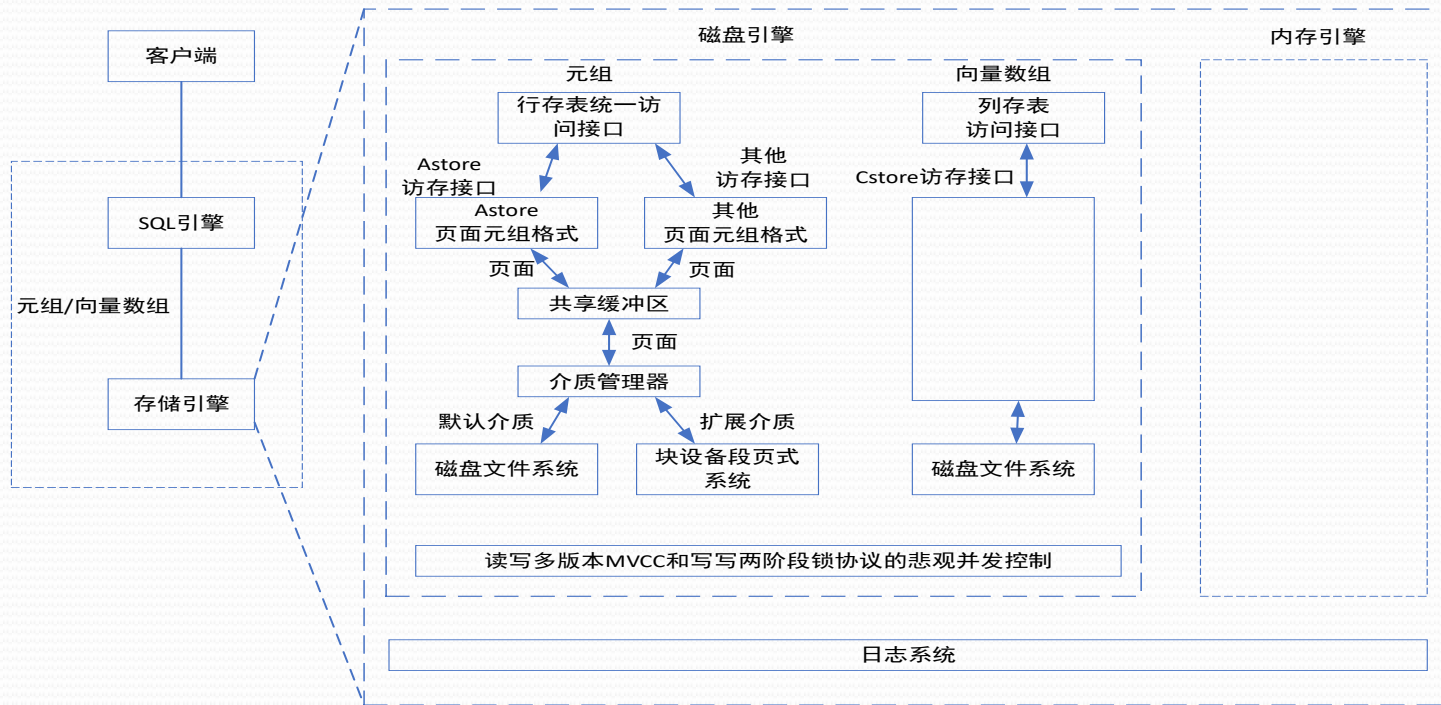
	R-LOCK	W-LOCK	C-LOCK
R-LOCK	Y	Y	N
W-LOCK	Y	N	N
C-LOCK	N	N	N

MV2PL混合协议

- 读锁和写锁相容。
- 写操作生成一个新的版本，其他事务的读操作**读取旧**的版本。
- **写事务提交时**要先获取加了写锁的数据对象的**验证锁**。由于验证锁和读锁不相容，导致写事务延迟提交，**直至加了写锁的数据对象被其他读事务释放**。
- 一旦写事务获得验证锁，旧版本即可丢弃，写事务提交并释放验证锁。
- 系统最多维护数据对象的两个版本。
- Oracle采用MV2PL。

国产数据库——openGauss

存储引擎之并发控制（拓展了解）



对于读、写并发操作，采用多版本并发控制（MVCC，multi-version concurrency control）；对于写、写并发操作，采用基于两阶段锁协议（2PL，two-phase locking）的悲观并发控制（PCC，pessimistic concurrency control）。



天猫“双11” 淘宝网

每秒订单创建峰值达到 49.1万笔



春运 中国铁路12306网站

日售票峰值达到1381.8万张

思考：

找任务的关键点

- 三级封锁协议和2PL协议思路有何不同？
- ANSI/ISO SQL将事务的隔离级别（也就是对事务并发控制的等级）分为读未提交（**READ UNCOMMITTED**）、读已提交（**READ COMMITTED**）、可重复读（**REPEATABLE READ**）、串行化（**SERIALIZABLE**）四个等级，教材到此为止的方法和原理如何与之关联？