

事务并发控制算法(一)



胡卉芪
华东师范大学
数据科学与工程学院
hqhu@dase.ecnu.edu.cn



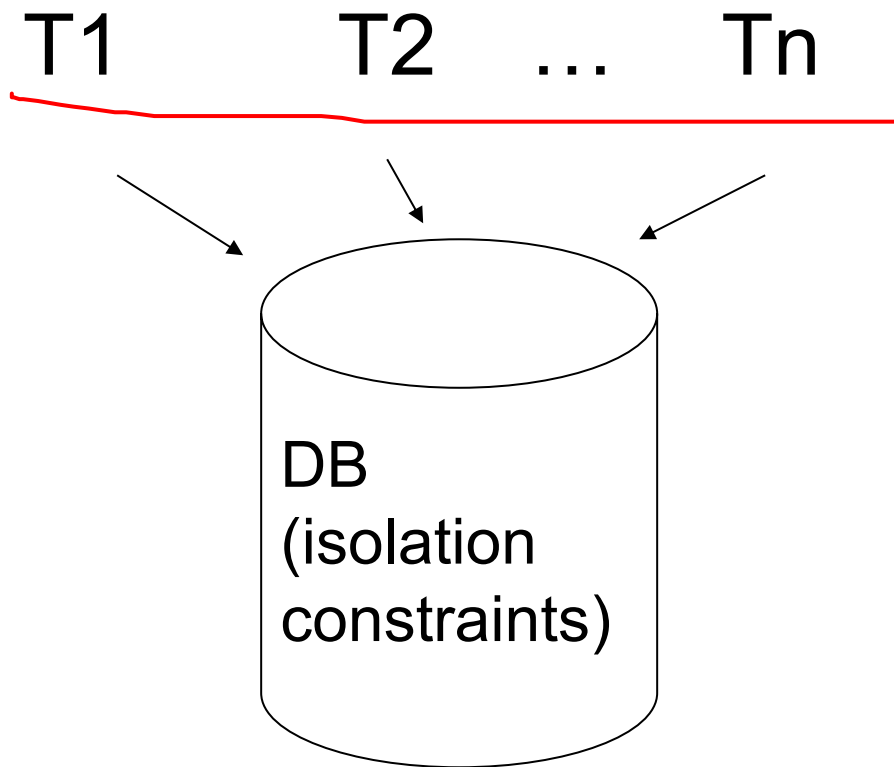
Revisit

- 串行调度
- 可串行化
 - 调度的结果和某个串行调度结果等价
- 冲突可串行化
 - 所有冲突操作的偏序关系一致

Revisit并发控制

- 并发控制可以看做一套算法
 - 输入：多个事务
 - 输出：满足一定隔离级别的调度

RC



并发控制算法的分类

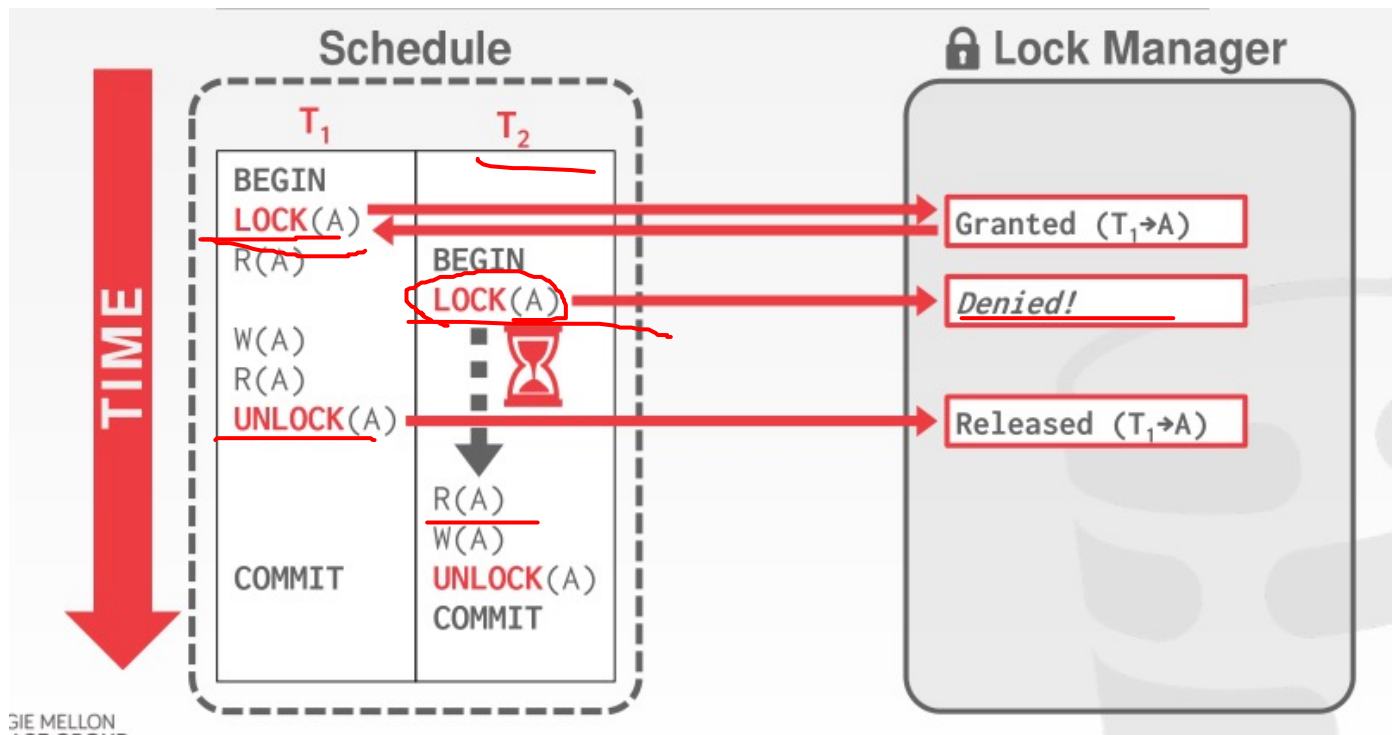
- 悲观/乐观
 - 悲观：2PL
 - 乐观：基于时间戳的并发控制(Timestamp ordering TO)
，乐观并发控制(OCC)
- 基于锁/基于时间戳
 - 基于锁：2PL
 - 基于时间戳：TO, OCC
- 并发控制的关键
 - 如何检测冲突
 - 如何处理冲突



2-Phase Locking

2PL

- 使用锁实现并发控制



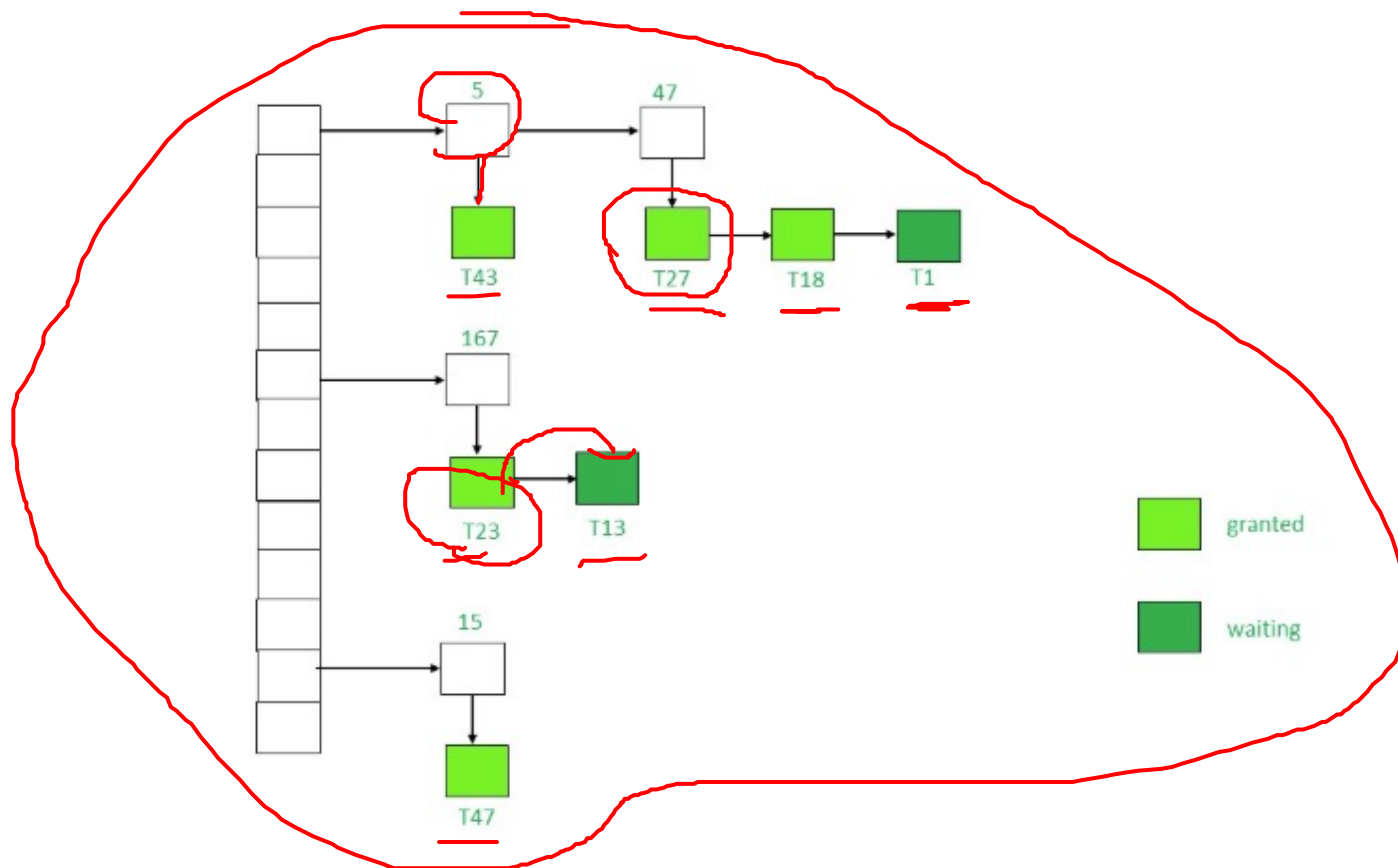
2PL

- 基本锁类型
 - 共享锁S-Lock，互斥锁X-Lock

Compatibility Matrix		
	Shared	Exclusive
Shared	<u>✓</u>	<u>X</u>
Exclusive	<u>X</u>	<u>X</u>

锁表

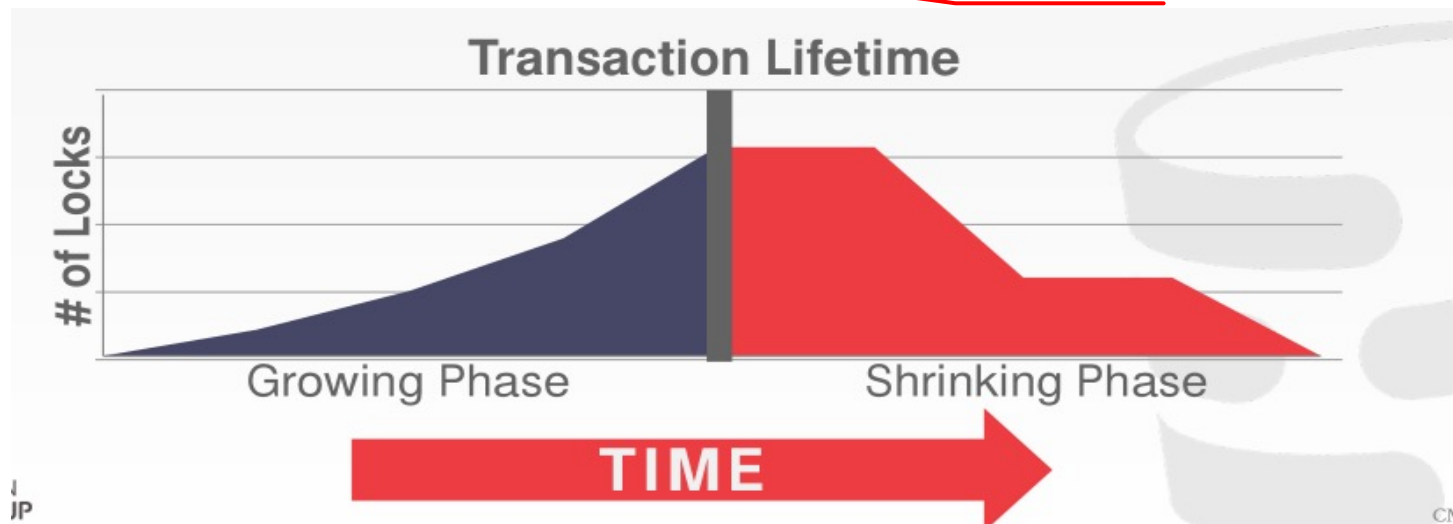
- 每个数据项上都有授权锁和等待锁的事务列表



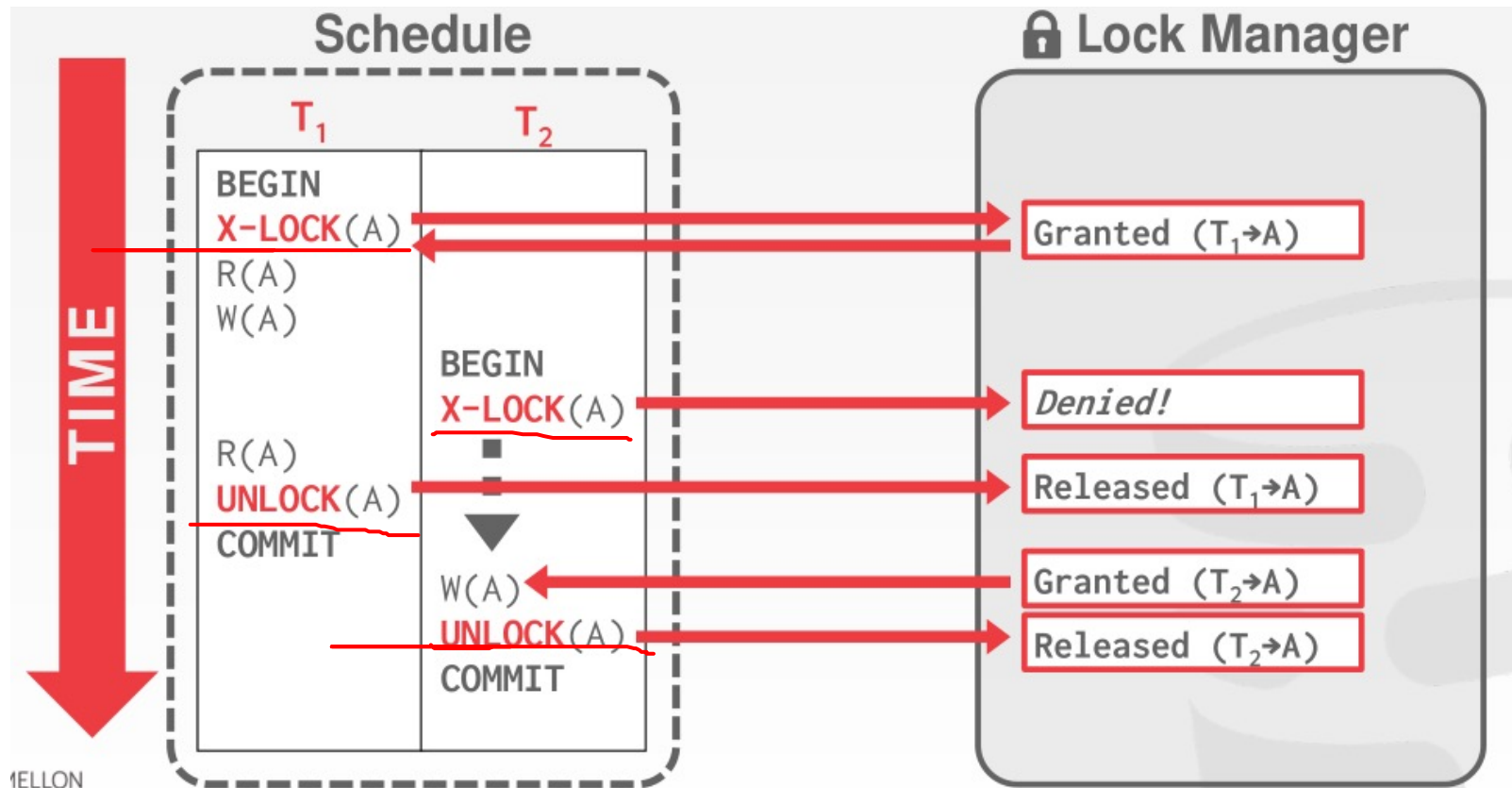
2PL

- Phase-1: Growing
 - 对读写操作申请锁
- Phase-2: Shrinking
 - 释放锁，不允许申请新锁

对同一个事务而言！



2PL例子



1ELLON



2PL例子

- 能保证冲突可串行化
 - 2PL产生调度属于冲突可串行化调度的子集，也意味着牺牲了一些并发性
- 问题
 - 级联回滚
 - 死锁



2PL属于冲突可串行化的原因

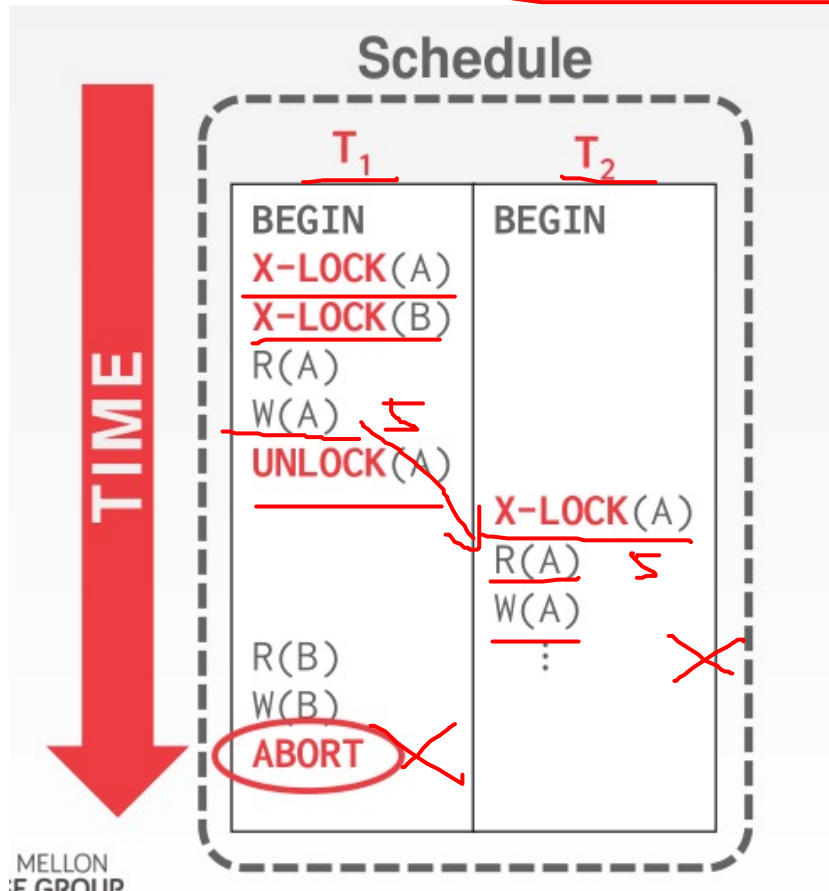
- 假设有 n 个事务 $T_1, T_2 \dots T_n$. 设 T_i 是其中有第一个解锁的事务, 如 $U_i(X)$. 那么将 T_i 所有动作移动到其他 $n-1$ 个事务之前是可能的
 - 简单来说可以看做第一个解锁操作决定了调度顺序

原因: 反证法

- 考虑 T_i 与 T_j 中的冲突操作 $W_i(Y), W_j(Y)$. 如果 $W_j(Y)$ 在前, 可能序列是:
 $T_j \rightarrow T_i$
 - $W_j(Y) \dots U_j(Y) \dots L_i(Y) \dots W_i(Y)$
- 由于 T_i 是第一个解锁的, 所以 $U_i(X)$ 在 $U_j(Y)$ 之前
 - $W_j(Y) \dots U_i(X) \dots U_j(Y) \dots L_i(Y) \dots W_i(Y)$ --这不是2PL产生的调度

2PL级联回滚

- T1回滚后T2必须回滚



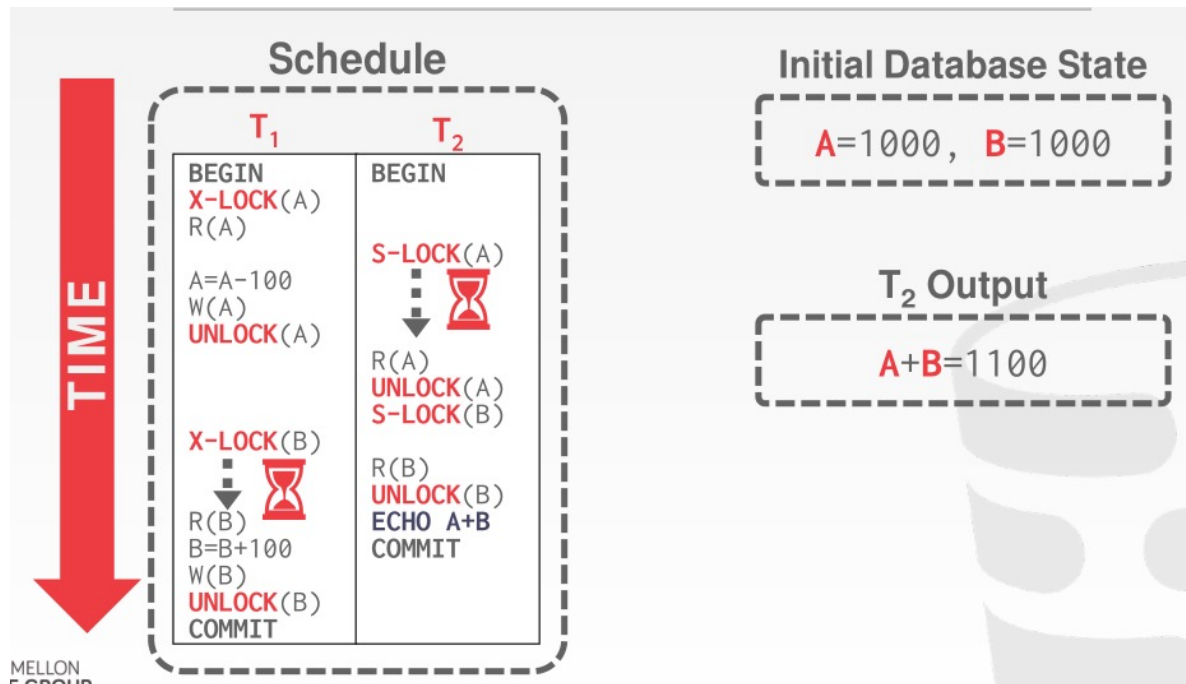
Strict 2PL (S2PL)

- 事务commit或者abort之后才释放锁
 - 为何没有级联回滚?
- Abort操作：需要释放锁

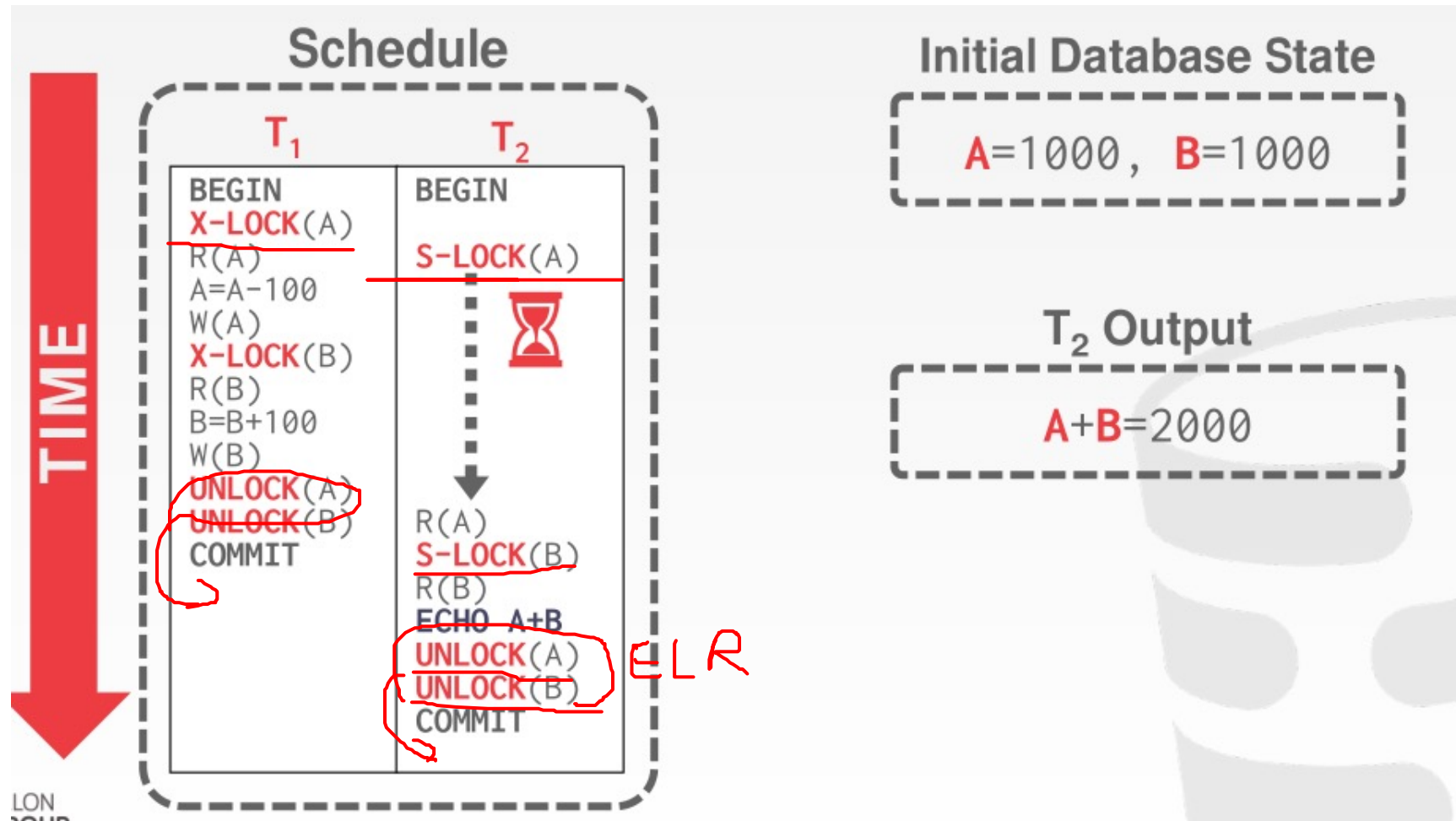


S2PL例子

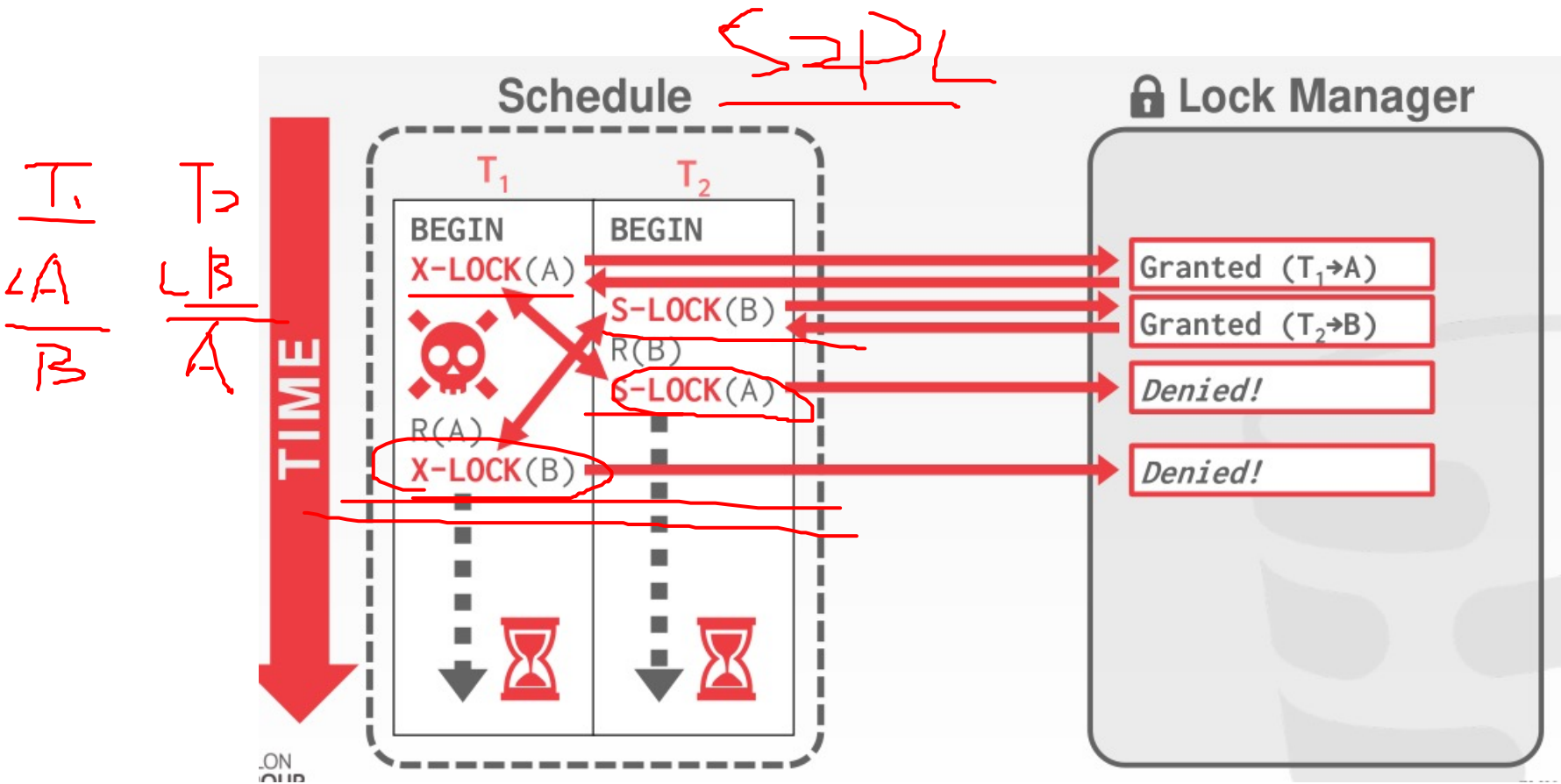
- 不用2PL例子：A转100给B
 - 结合上节课这属于什么异常？



S2PL例子



2PL的死锁问题

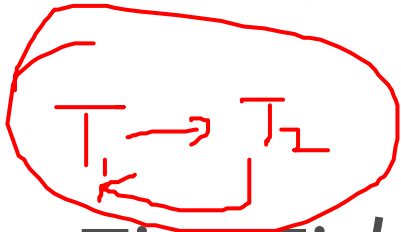


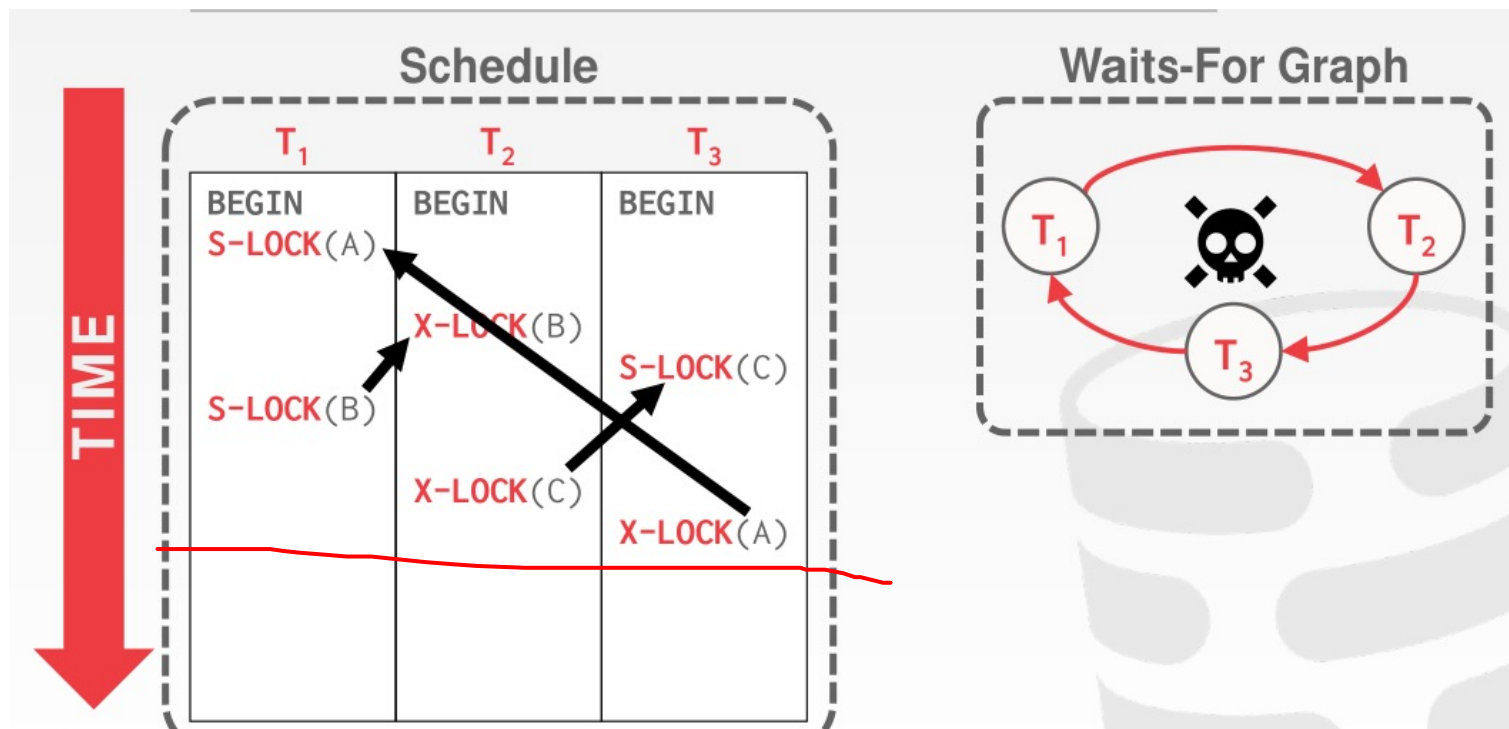
解决死锁

- 超时检测
- 死锁检查
- 死锁预防
 - Wait-die
 - ~~Wound-die~~
- 元素排序
- ...



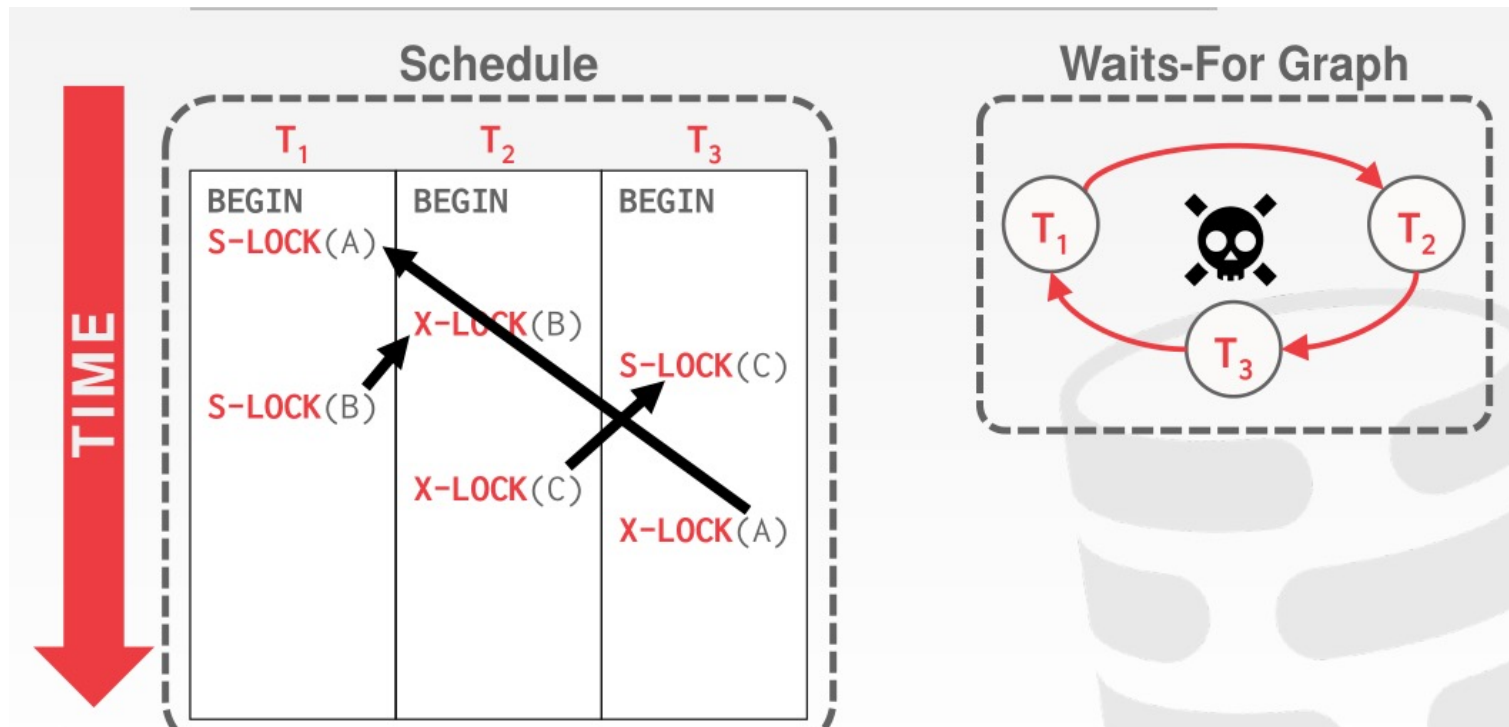
死锁检测

- 锁等待图 
 - 有向图， $T_i \rightarrow T_j$ 如果 T_i 在等 T_j 放锁
 - 死锁形成环，选择进行abort



死锁检测

- 锁等待图
 - 有向图， $T_i \rightarrow T_j$ 如果 T_i 在等 T_j 放锁
 - 死锁形成环，选择进行abort



元素排序

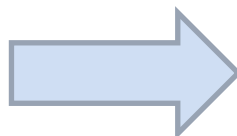
- 如果所有事务都按照元素的某个固定顺序去请求锁，则不会出现死锁。

$T_1: l_1(A); r_1(A); l_1(B); w_1(B); u_1(A); u_1(B);$

$T_2: l_2(C); r_2(C); l_2(A); w_2(A); u_2(C); u_2(A);$

$T_3: l_3(B); r_3(B); l_3(C); w_3(C); u_3(B); u_3(C);$

$T_4: l_4(D); r_4(D); l_4(A); w_4(A); u_4(D); u_4(A);$



$T_1: l_1(A); r_1(A); l_1(B); w_1(B); u_1(A); u_1(B);$

$T_2: l_2(A); l_2(C); r_2(C); w_2(A); u_2(C); u_2(A);$

$T_3: l_3(B); r_3(B); l_3(C); w_3(C); u_3(B); u_3(C);$

$T_4: l_4(A); l_4(D); r_4(D); w_4(A); u_4(D); u_4(A);$

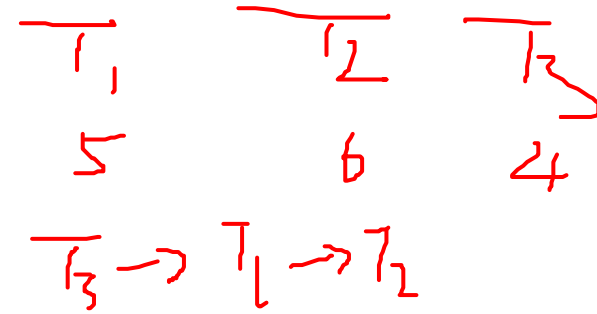
T_1 T_2
 A ~~A~~ A
 B ~~A~~ B

	T_1	T_2	T_3	T_4
1)	$l_1(A); r_1(A);$			
2)		$l_2(A);$ 被拒绝		
3)			$l_3(B); r_3(B);$	
4)				$l_4(A);$ 被拒绝
5)			$l_3(C); w_3(C);$	
6)			$u_3(B); u_3(C);$	
7)	$l_1(B); w_1(B);$			
8)	$u_1(A); u_1(B);$			
9)		$l_2(A); l_2(C);$		
10)		$r_2(C); w_2(A);$		
11)		$u_2(A); u_2(C);$		
12)				$l_4(A); l_4(D);$
13)				$r_4(D); w_4(A);$
14)				$u_4(A); u_4(D);$

Timestamp Ordering



时间戳



- 用时间戳来表现事务的调度顺序
 - 如果 $TS(T_i) < TS(T_j)$ 那么可以认为在可串行化调度中 T_i 在 T_j 前调度
- 时间戳的形式
 - 物理时钟 ✗
 - 逻辑计数器
 - 单调递增计数
- 分配时机
 - 不同算法不同

Handwritten notes for allocation timing:

T_0
266



TO的想法

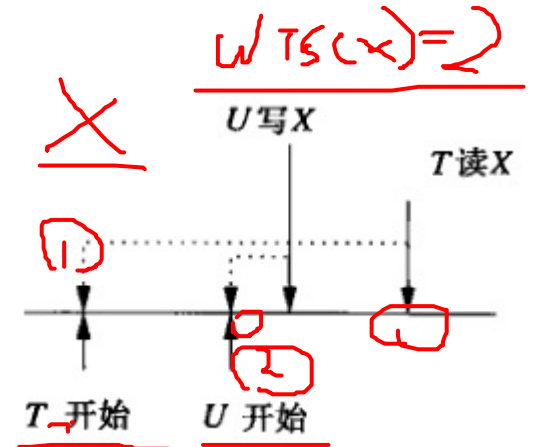
- 在事务开始时为每个事务分配时间戳，
决定调度顺序
 - 如何检测冲突？
 - 对每个操作，如何数据访问来自未来操作，那么是有冲突的
 - 如何解决冲突？
 - Abort冲突事务，重启(分配新的时间戳)

TO算法的想法

- 事务执行时不使用锁
- 每个数据X都带两个时间戳
 - W-TS(X): 最近成功写X的事务时间戳
 - R-TS(X): 最近成功读X的事务的时间戳

TO基本规则-读操作

- 对一个事务 T_i , 假设它读取数据 X , 如果 $TS(T_i) < W-TS(X)$:
 - 意味着读到了未来的数据
 - 事务 T_i abort & restart
- 否则
 - 执行读操作, 并更新 $R-TS(X)$



TO基本规则-写操作

- 对一个事务 T_i , 假设它写数据 X , 如果 $TS(T_i) < R-TS(X)$ 或者 $TS(T_i) < W-TS(X)$:

- 意味未来操作跳过了 T_i
- 事务 T_i abort & restart

- 否则

- 执行写操作, 并更新 $W-TS(X)$

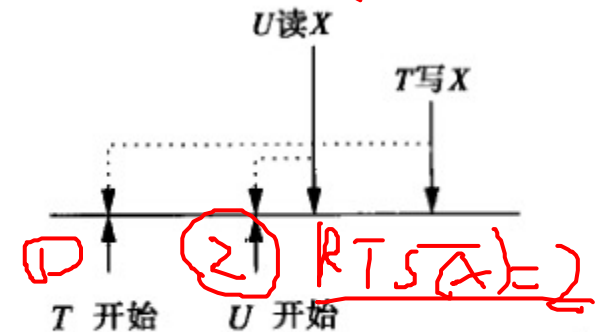
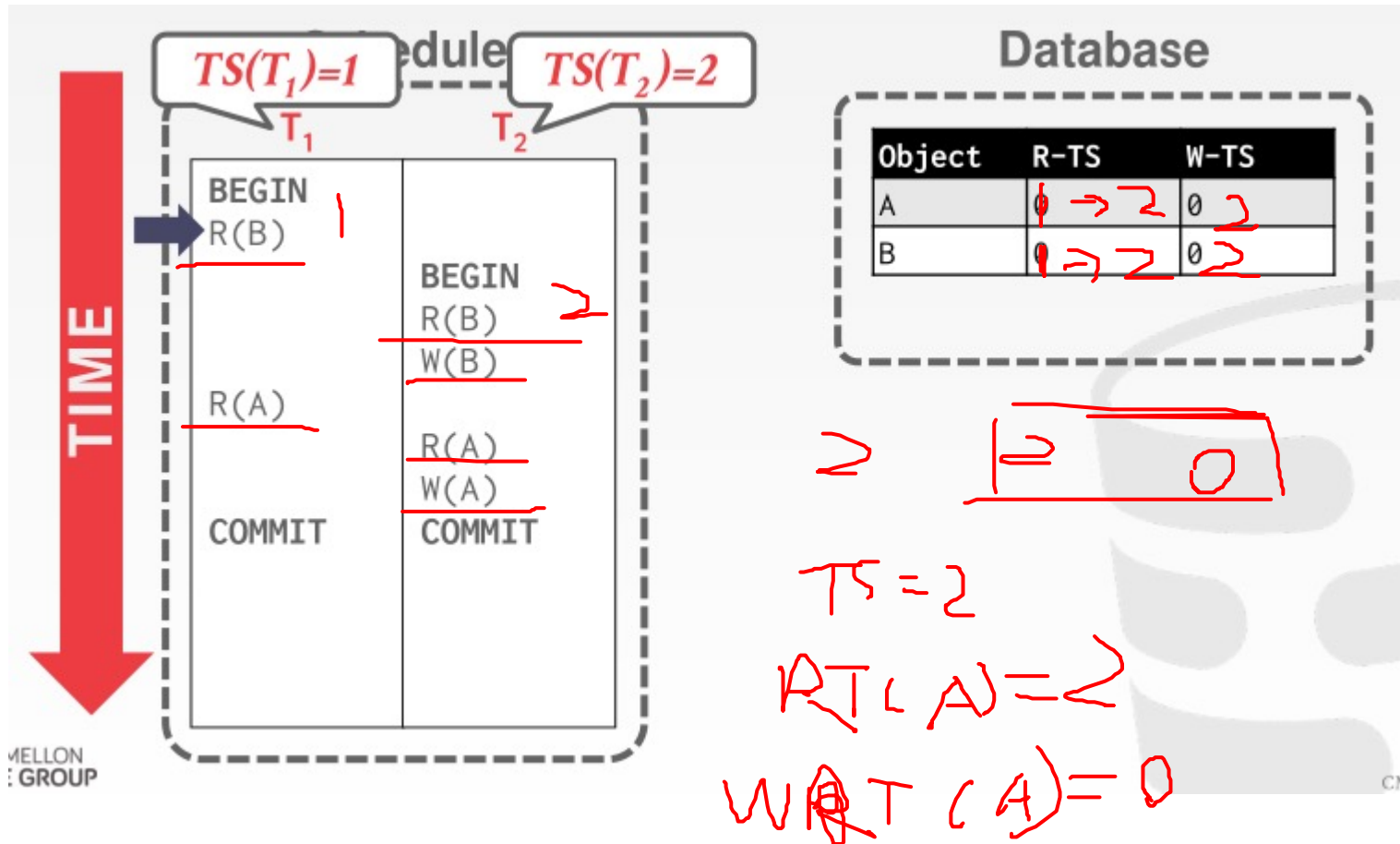
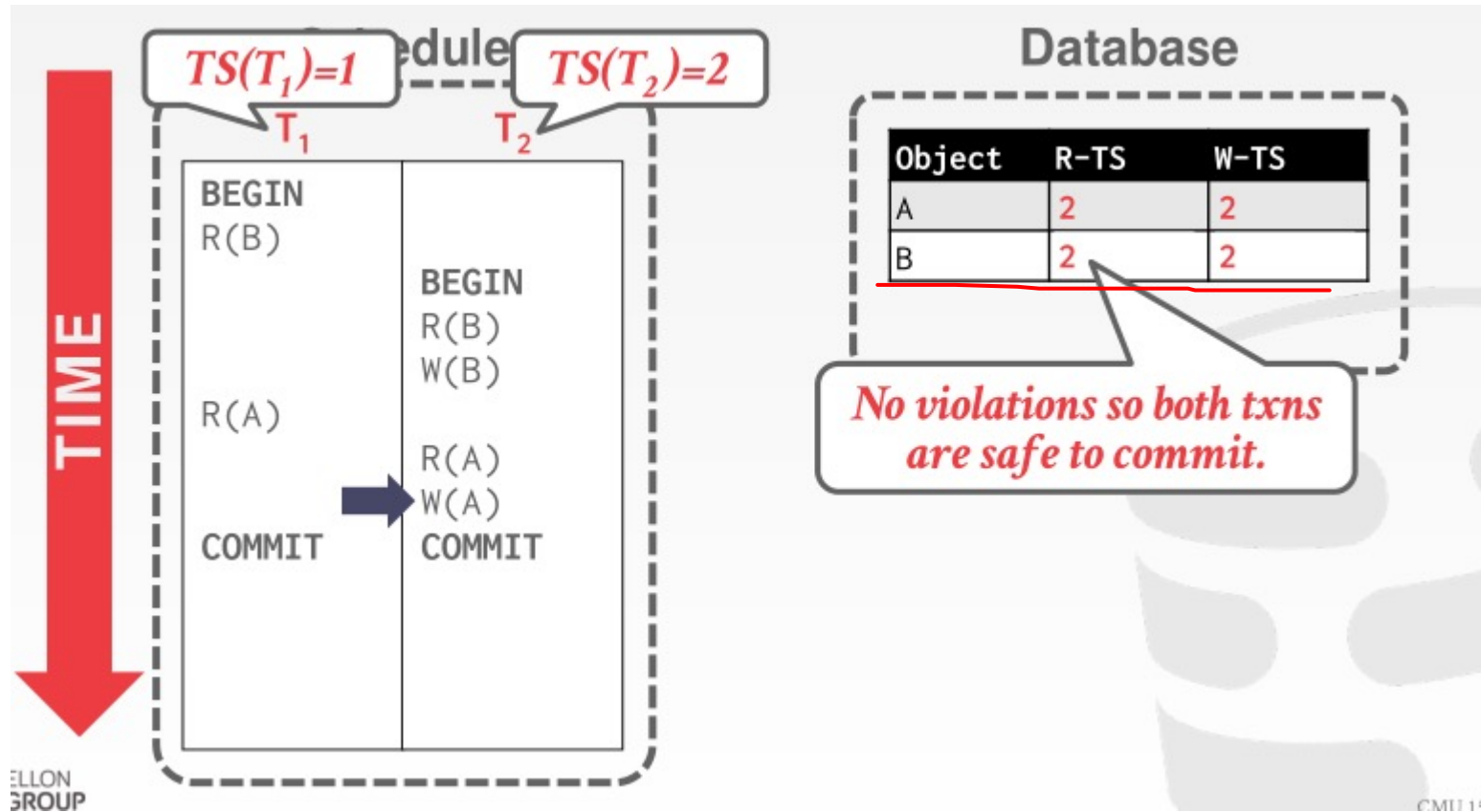


图 7-36 事务 T 试图做过晚的读

TO基本规则示例

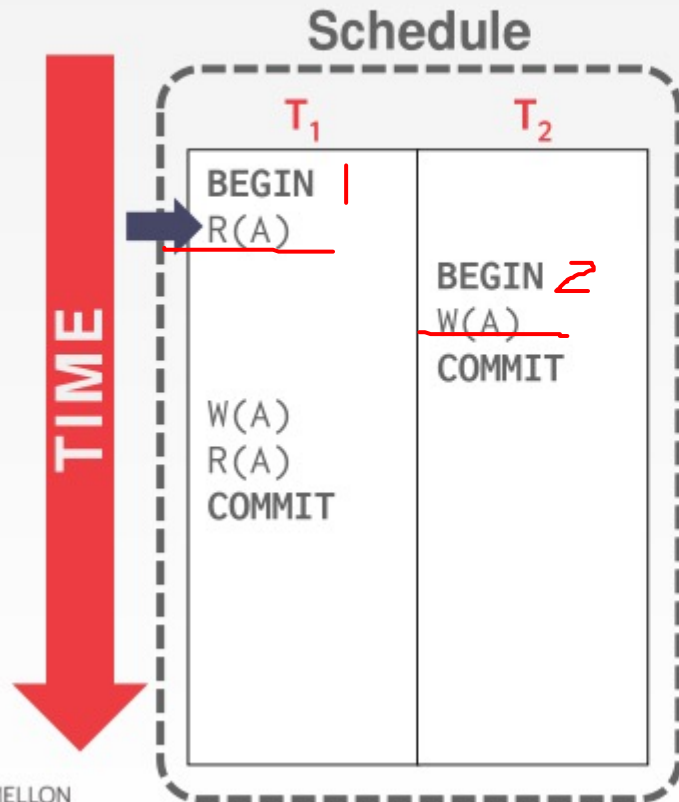


TO基本规则示例



TO基本规则示例

BASIC T/O – EXAMPLE #2



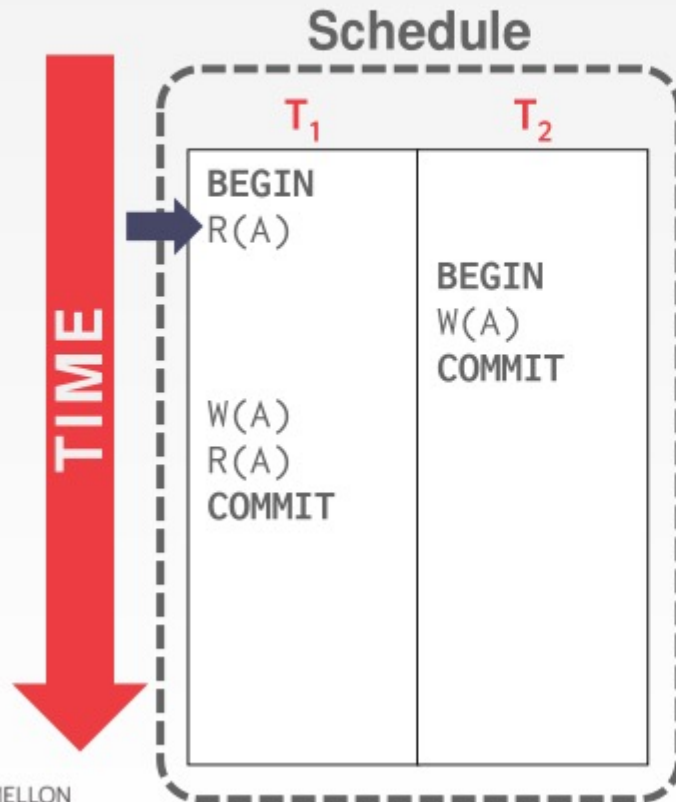
Database

Object	R-TS	W-TS
A	1	0
B	0	0



TO基本规则示例

BASIC T/O – EXAMPLE #2

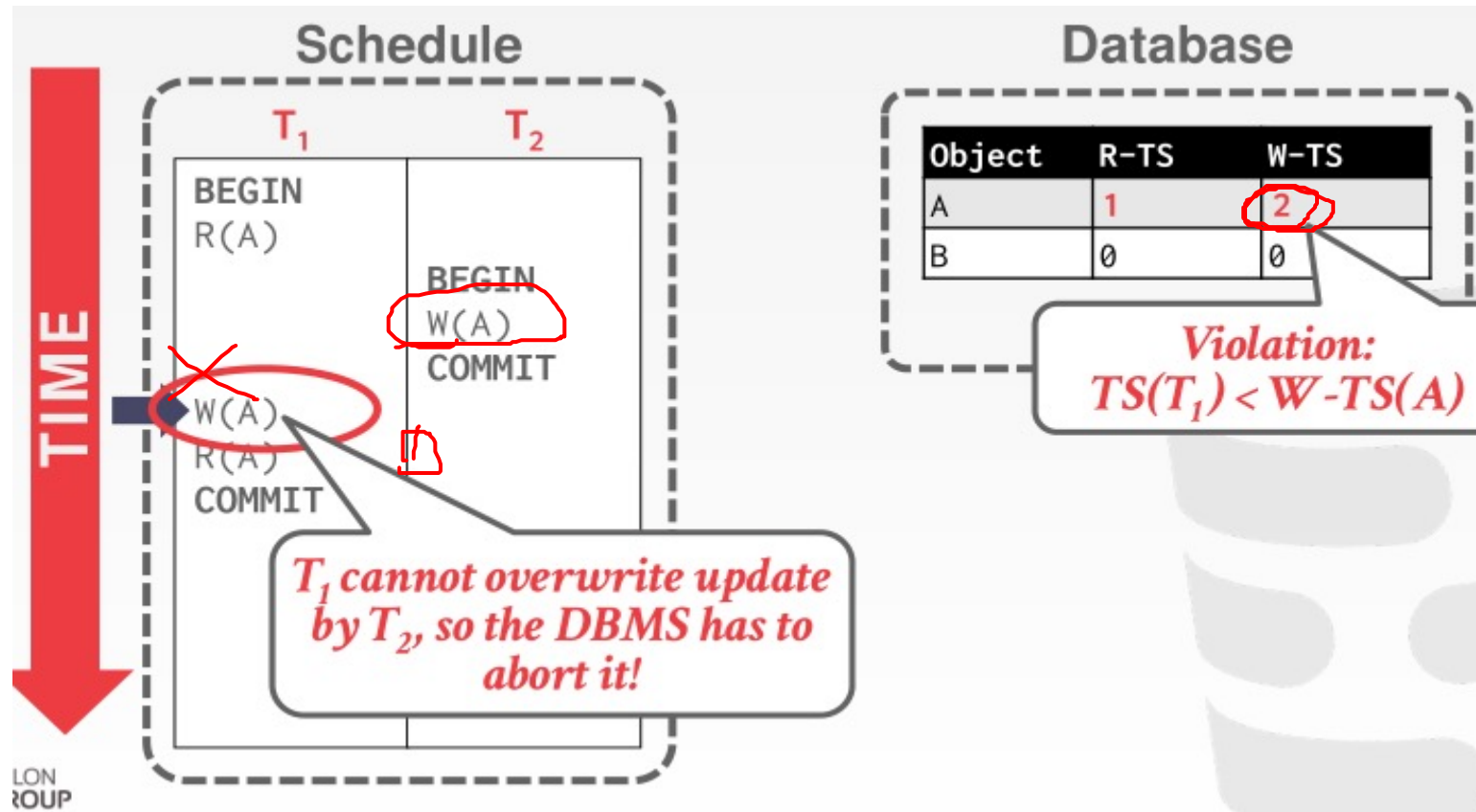


Database

Object	R-TS	W-TS
A	1	0
B	0	0



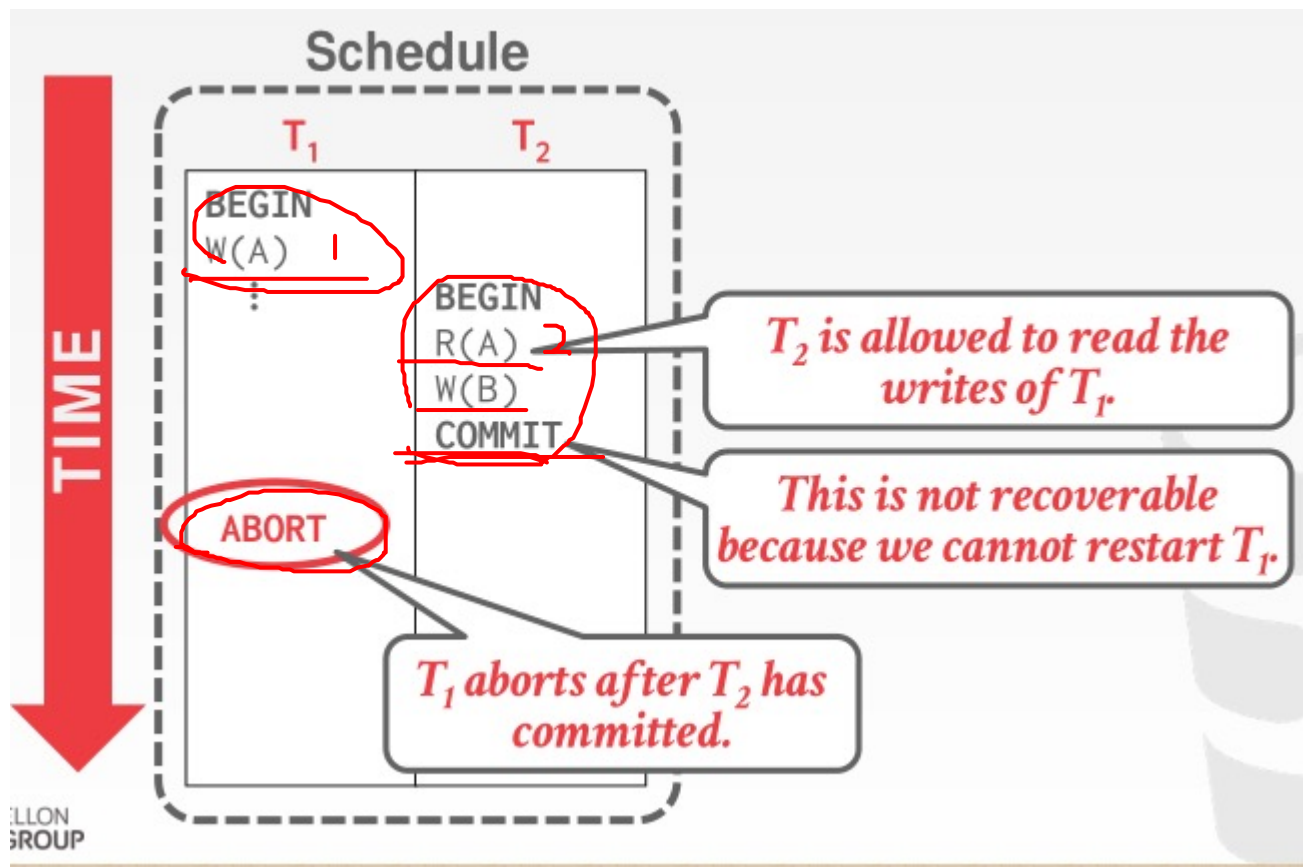
TO基本规则示例



2/1

运用基本规则的问题

- 事务回滚->脏读->级联回滚



解决脏读和级联回滚

- 对每个数据项X再设置一个提交位 $C(X)$, 只有 $C(X)$ 为真时才允许读



解决脏读的TO算法规则

- 对X的读请求
 - 如果 $TS(T) \geq WT(X)$
 - 如果C(X)为真，同意请求，并更新RS(X)
 - 如果C(X)为假，推迟T直到C(X)为真或写X的事务终止
 - 否则
 - Abort & 重启

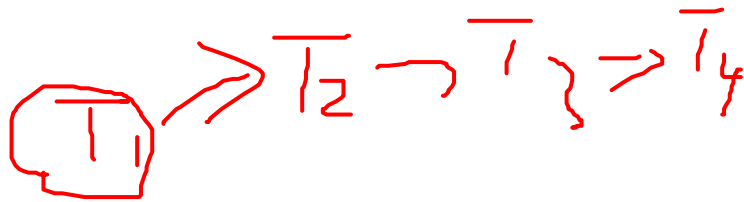
解决脏读的TO算法规则

- 对X的写请求
 - 如果 $TS(T) \geq WT(X) \ \&\& \ TS(T) \geq RT(X)$
 - 置 $WT(X) = TS(T)$, $C(X) = false$;
 - 否则
 - Abort & restart



解决脏读的TO算法规则

- Commit
 - 置所有C(X)为真
- Abort
 - 写数据撤销



解决脏读的TO算法规则

- 实例

T_1	T_2	T_3	A	B	C
200	150	175	RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
<u>$r_1(B);$</u>	<u>$r_2(A);$</u>	<u>$r_3(C);$</u>	RT=150	RT=200	RT=175
<u>$w_1(B);$</u>				WT=200	
<u>$w_1(A);$</u>	150		WT=200		
	<u>$w_2(C);$</u>				
	中止				
		<u>$w_3(A);$</u>			

175

A $\overline{T_3}$ $\overline{T_1}$
 $\overline{w_2(A)}$ $\overline{w_1(A)}$