第12讲 事务及并发控制

什么是事务

- [Definition] 事务(Transaction) 事务是数据库管理系统提供的控制数据操作的一种手段，通过这一手段，应用程序员将**一系列的数据库操作**组合在一起作为**一个整体**进行操作和控制，以便数据库管理系统能够提供一致性状态转换的保证。

例如：“银行转帐”事务T：从帐户A过户5000元到帐户B上

```
read(A);
A := A - 5000;
write(A);
read(B);
B := B + 5000;
write(B);
Commit;
```

注：read(X)是从数据库传送数据项X到事务的工作区中；write(X)是从事务的工作区中将数据项X写回数据库。

事务的提交(commit)：事务完成，所有操作全部成功执行，数据库进入新状态。
事务的回滚(rollback)：事务退出，所有已完成操作撤销，数据库恢复到开始前状态。

什么是事务

事务的特性

事务的特性: ACID

原子性Atomicity : DBMS能够保证事务的一组更新操作是原子不可分的，即对DB而言，要么全做，要么全不做（不能只更新A的账户不更新B的账户）。

一致性Consistency: DBMS保证事务操作前后，数据库均符合一致性的约束（例如主键约束、外键约束、非空约束，自定义约束等）。（转账后A和B的总钱数与之前必须相等，钱数不能小于0等）

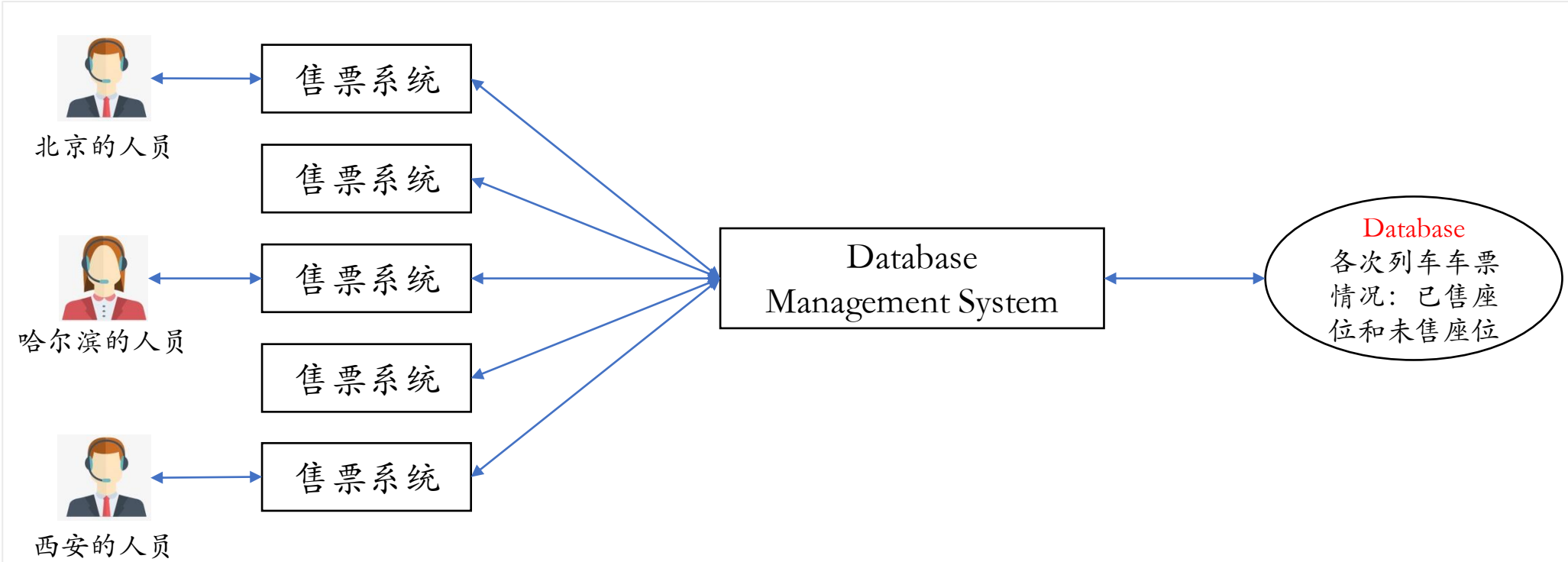
隔离性Isolation: DBMS保证并发执行的多个事务之间互相不受影响。例如两个事务T1和T2, 即使并发执行，也相当于或者先执行了T1,再执行T2;反之亦然。（AB同时给C转账不会造成C账户余额错误）

持久性Durability: DBMS保证已提交事务对数据库的改变是永久的，即使数据库出现故障。（A给B转账已完成，即使银行数据库断电，也不会影响这笔转账了。）

为什么要进行并发控制

多用户同时在数据库上执行事务，保证不会出现冲突和数据不一致：

- 火车票售票系统：避免超售、座位冲突；购票与退票正确处理，金额正确
- 在线购物平台、选课系统
- 并发可以提高系统吞吐率；减少等待时间。



事务调度与可串行性

- [Definition]事务调度(schedule)：一组事务的基本步(读、写、其他控制操作如加锁、解锁等)的一种执行顺序称为对这组事务的一个调度。
- 并发(或并行)调度：将一组同时进行的事务中的各个操作以某种顺序来执行。

S	T_1	T_2
1	Read A	
2	A=A-10	
3	Write A	
4	Read B	
5	B=B+10	
6	Write B	
7		Read B
8		B=B-20
9		Write B
10		Read C
11		C=C+20
12		Write C

串行调度

S	T_1	T_2
1	Read A	
2		Read B
3	A=A-10	
4		B=B-20
5	Write A	
6		Write B
7	Read B	
8		Read C
9	B=B+10	
10		C=C+20
11	Write B	
12		Write C

S	T_1	T_2
1	Read A	
2	A=A-10	
3		Read B
4	Write A	
5		B=B-20
6	Read B	
7		Write B
8	B=B+10	
9		Read C
10	Write B	
		C=C+20
		Write C

并发调度

一种简单的事务调度的标记模型

表达事务调度的一种模型

$r_T(A)$: 事务T读A。　　$w_T(A)$: 事务T写A

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

T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

A，B是数据库中的对象，例如元组值、属性值等。这里只考虑读写操作。读写之间可能有其他内存中的计算步骤。

事务调度与可串行性

基本概念

并发调度的正确性：当且仅当在这个并发调度下所得到的新数据库结果与分别串行地运行这些事务所得的新数据库完全一致，则说调度是正确的。

[Definition]**可串行性**：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(Serializable)或具有可串行性(Serializability)。

事务调度与可串行性

冲突可串行性（Conflict Serializability）

- 冲突：调度中针对同一元素的两个连续操作，分别属于不同事务，其中含有写操作。
- 有冲突的两个操作是不能交换顺序的，可能会该变事务执行结果
- 没有冲突的两个操作是可交换顺序的，不改变结果
- 注：同一事务的操作顺序是固定的。可理解为同一事务中所有操作均互相“冲突”

- **不同事务对同一元素的两个写操作是冲突的（写-写冲突）**

$$w_i(X); w_j(X)$$

- **不同事务对同一元素的一读一写操作是冲突的（读-写或写-读冲突）**

$$w_i(X); r_j(X) \qquad r_i(X); w_j(X)$$

数据一致性问题和冲突

数据不一致的情况：

1. 丢失修改：修改过后的数据未能保存，而是被其他事务覆盖了（写-写冲突）

2. 不能重复读：两次读数据不一致，中间被写了（读-写冲突）

3. 脏读：读取了没有提交的修改数据 （写-读冲突）

1. 丢失修改

T1	T2
Read A (DB：A=50 M: A=50)	
	Read A (DB：A=50 M: A=50)
Update A (设 A=A-1 M: A=49)	
	Update A (设 A=A-2 M: A=48
Write A (M: A=49 (DB：A=49)	
	WriteA(A_2) (M: A=48 DB：A=48

A 被修改了 2 次，但后一次修改覆盖了前一次修改。从而丢失了 A 的累积修改结果。

2. 不能重复读

T1	T2
	Read A (DB：A=A1 M: A=A1)
	Update A (M: A=A2)
Read A (DB：A=A1 M: A=A1)	
	Write A (M: A=A2 DB：A=A2)
Read A (DB：A=A2 M: A=A2)	

$A_1 \neq A_2$ 两次读的不是同一数据。

3. 脏读

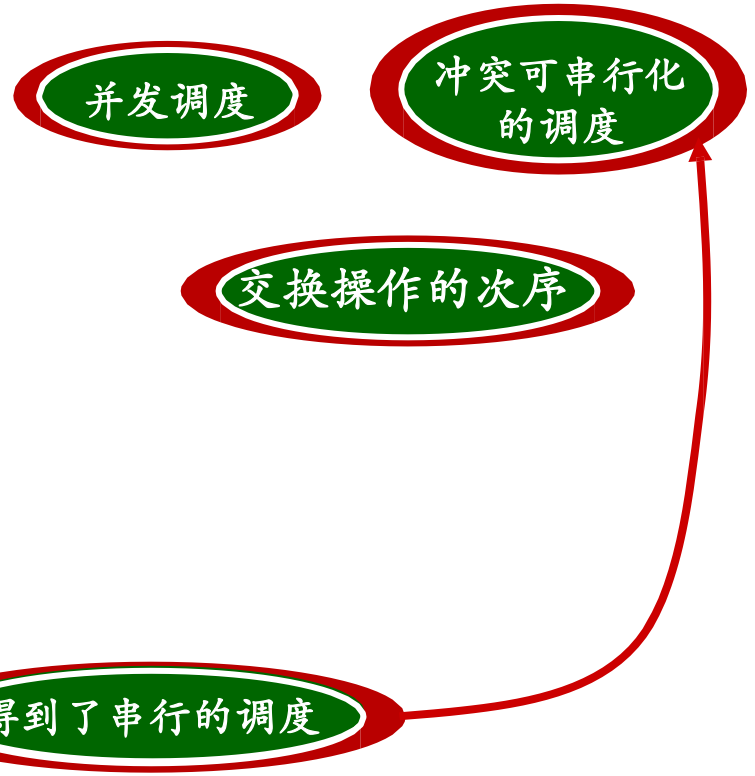
T1	T2
Read A (DB：A=A1 M: A=A1)	Read A (DB：A=A1 M: A=A1)
	Update A (M: A=A2)
	Write A (M: A=A2 DB：A=A2)
Read A (DB：A=A2 M: A=A2)	
	Roll Back (DB：A=A1)
Read A (M: A=A2)	

A_2 已无效，应为 A_1。

事务调度和可串行性

冲突可串行性

冲突可串行性： 一个调度，如果通过交换两个相邻的无冲突的操作顺序能够转换到某一个串行的调度，则称此调度为冲突可串行化的调度。



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

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

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

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

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

事务调度和可串行性

冲突可串行性

- 冲突可串行性 是比可串行性 要严格的概念
- 满足冲突可串行性，一定满足可串行性；反之不然。
- 并发调度的正确性 ⊇ 可串行性 ⊇ 冲突可串行性

可串行化调度

$w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$

不是冲突可串行化调度

$w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$

等效（结果都是由$W_3(X)$和$W_2(Y)$决定）

但不能无冲突转换:
(1) 中对X写操作的顺序是: W_2, W_1, W_3
(2) 中对X写操作的顺序是: W_1, W_2, W_3

冲突可串行性判别算法

算法表达

如何判断一个调度是冲突可串行性的?

冲突可串行性判别算法

- 构造一个前驱图(有向图)
- 结点是每一个事务T_i。如果T_i的一个操作与T_j的一个操作发生冲突，且T_i在T_j前执行，则绘制一条边，由T_i指向T_j, 表征T_i要在T_j前执行。
- 测试检查: 如果此有向图没有环，则是冲突可串行化的!

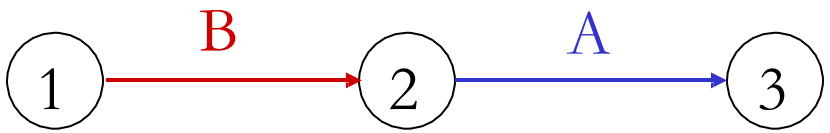
冲突可串行性判别算法

示例

示例

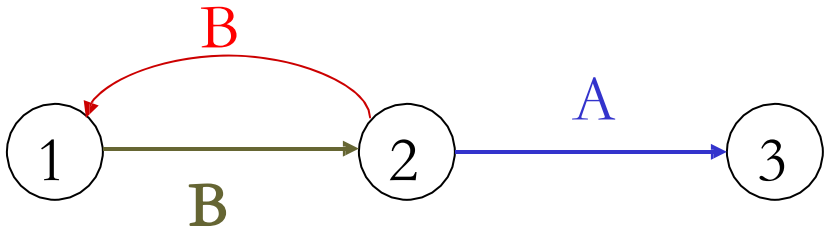
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

冲突可串行化调度



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

非冲突可串行化调度



基于封锁的并发控制方法

基于封锁的并发控制方法

什么是锁?

“锁” 是控制并发的一种手段

- 全局变量，表示对特定元素进行特定操作的“权限”
- 每一事务对一个元素进行操作前，要获得对应的锁。
- 如果元素已被其他事务加锁，且无法再加锁，则需要等待其他事务解锁。
- 操作完成后要释放锁。事务结束后释放所有持有的锁。

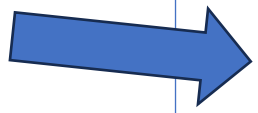
最简单的锁：每个数据元素只有唯一一个锁，任何操作都要先获得该锁，且不可共享。

$L_i(A)$ ：事务T_i对数据元素A加锁

$U_i(A)$ ：事务T_i对数据元素A解锁

基于封锁的并发控制方法

什么是锁？ 调度器可利用锁来实现（但不保证达到）冲突可串行性



T1	T2
$L_1(A)$; READ(A, t)	
t := t+100	
WRITE(A, t);	
$U_1(A)$	
$L_1(B)$	
	$L_2(A)$; READ(A,s)
	s := s*2; WRITE(A,s);
	$U_2(A)$;
	$L_2(B)$; “拒绝…”
READ(B, t)	
t := t+100	
WRITE(B,t);	
$U_1(B)$;	
	…获得“锁”
	READ(B,s)
	s := s*2; WRITE(B,s);
	$U_2(B)$;

T1	T2
$L_1(A)$; READ(A, t)	
t := t+100	
WRITE(A, t);	
$U_1(A)$	
$L_1(B)$	
READ(B, t) t := t+100	
WRITE(B,t);	
$U_1(B)$;	
	$L_2(A)$; READ(A,s)
	s := s*2; WRITE(A,s);
	$U_2(A)$;
	$L_2(B)$;
	READ(B,s)
	s := s*2; WRITE(B,s);
	$U_2(B)$;

基于封锁的并发控制方法

封锁协议需要考虑什么？

封锁协议之锁的类型

- **排他锁X** (eXclusive locks)

  加锁事务能对该元素读、写，其他任何事务都不能**对其加锁。**

- **共享锁S** (Shared locks)

  加锁事务能对该元素读，但不能对其写；可以和其他S锁共存。

- **更新锁U** (Update locks)

  初始读，以后可升级为写

如何利用不同类型的锁，既提高并发性，又保证一致性呢？

基于封锁的并发控制方法

封锁协议需要考虑什么？

封锁协议之相容性矩阵

如何表达封锁协议？

读锁写锁协议		申请的锁	
		S	X
持有锁的模式	S	是	否
	X	否	否

当某事务对一数据对象持有一种锁时，另一事务再申请对该对象加某一类型的锁，是允许(是)还是不允许(否)

更新锁协议		申请的锁		
		S	X	U
持有锁的模式	S	是	否	是
	X	否	否	否
	U	否	否	否

这只是简单形式，还有更丰富内容

基于封锁的并发控制方法

封锁协议需要考虑什么?

不同级别的加锁策略/时机：更高效 vs. 更好的数据一致性

1级协议(1-LP): Read uncommitted

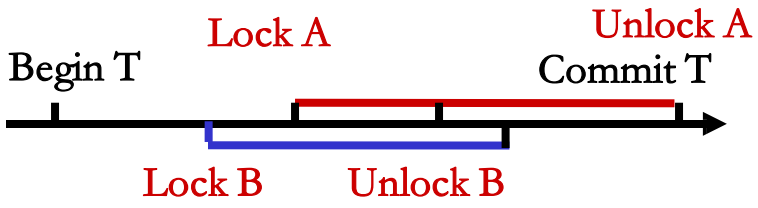
有写要求的数据对象A加排他锁X，事务提交时刻解锁。

Yes：防止丢失修改

No：重复读、脏读

读操作不需要加锁

2级协议(2-LP): Read committed

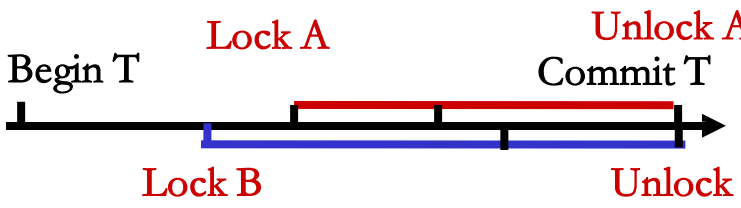


1级协议之上，对读数据B加共享锁S，不再访问后即刻解锁。

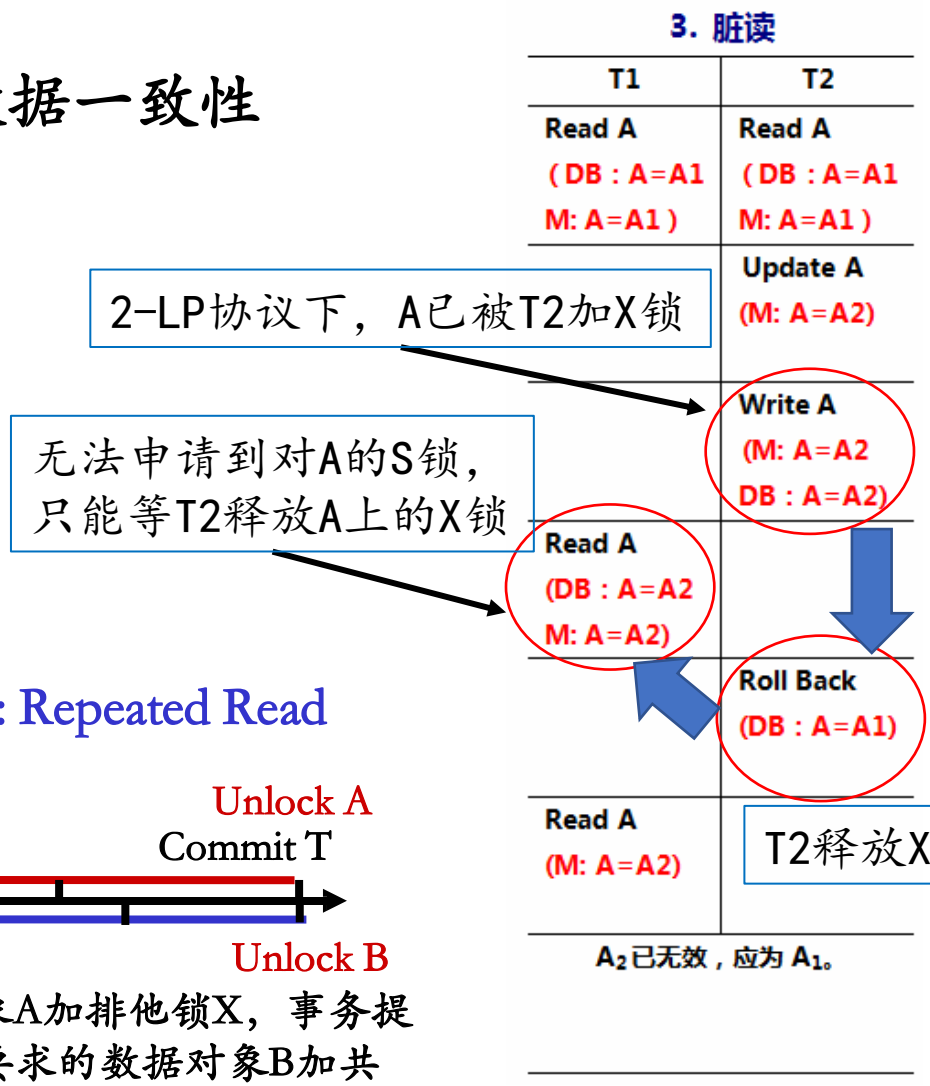
Yes：防止丢失修改，脏读；

No：重复读

3级协议(3-LP): Repeated Read



有写要求的数据对象A加排他锁X，事务提交时刻解锁。有读要求的数据对象B加共享锁S，事务提交时刻解锁。防止三种不一致性。



基于封锁的并发控制方法

封锁协议需要考虑什么?

不同级别的加锁策略/时机：更高效 vs. 更好的数据一致性

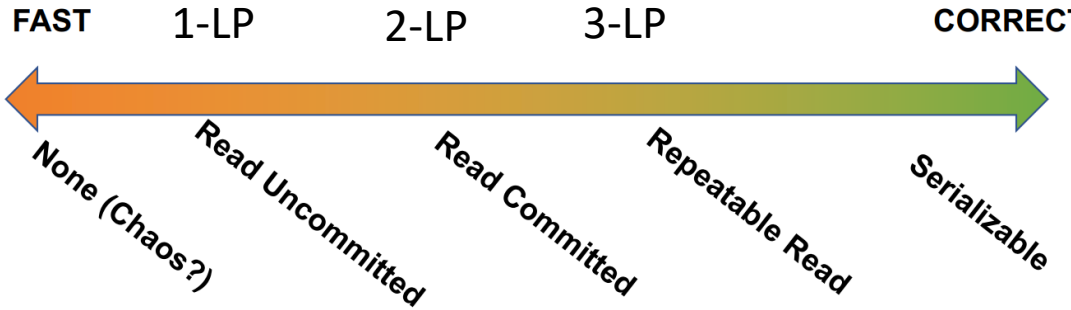
1级协议(1-LP): Read uncommitted

有写要求的数据对象A加排他锁X，事务提交时刻解锁。

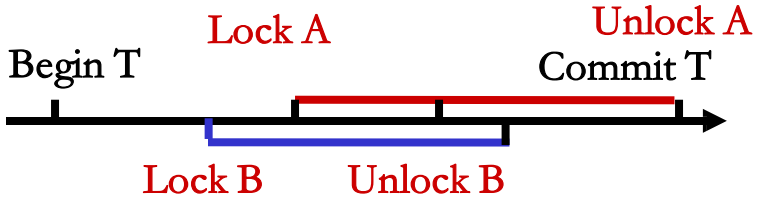
Yes：防止丢失修改

No：重复读、脏读

读操作不需要加锁



2级协议(2-LP): Read committed

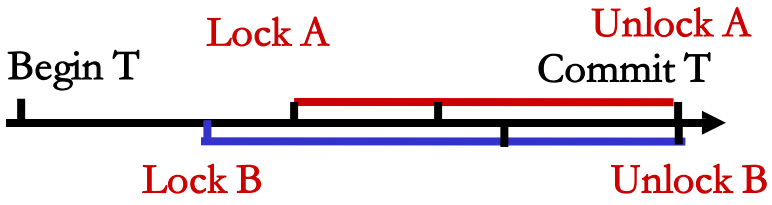


1级协议之上，对读数据B加共享锁S，不再访问后即刻解锁。

Yes：防止丢失修改，脏读；

No：重复读

3级协议(3-LP): Repeated Read



有写要求的数据对象A加排他锁X，事务提交时刻解锁。有读要求的数据对象B加共享锁S，事务提交时刻解锁。防止三种不一致性。

基于封锁的并发控制方法

封锁协议需要考虑什么？

封锁协议之封锁粒度(LOCKING GRANULARITY)

封锁粒度是指封锁数据对象的大小。

粒度单位：属性值➔元组➔元组集合➔整个关系➔整个DB

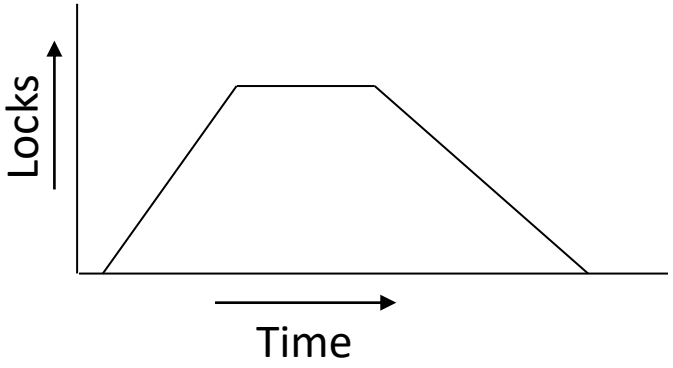
某索引项➔整个索引

由前往后：并发度小，封锁开销小；

由后往前：并发度大，封锁开销也大。

两段封锁协议 (2PL: Two-Phase Locking protocol)

- 读写数据之前要获得锁。每个事务中所有封锁请求先于任何一个解锁请求。
- 两阶段：加锁段，解锁段。加锁段中不能有解锁操作，解锁段中不能有加锁操作.
- 通过两阶段封锁协议，可以保证实现冲突可串行



T1	T2
$R_1(B)$	$R_2(B)$
B := B-50;	$R_2(A)$
$W_1(B)$	Display(A+B)
$R_1(A)$	
A := A+50;	
$W_1(A)$	

按两段锁协议加锁解锁

T1	T2
Lock-X(B)	Lock-S(B)
$R_1(B)$	$R_2(B)$
B := B-50;	Lock-S(A)
$W_1(B)$	$R_2(A)$
Lock-X(A)	Display(A+B)
Unlock(B)	Unlock(A)
$R_1(A)$	Unlock(B)
A := A+50;	
$W_1(A)$	
Unlock(A)	

加锁阶段

解锁阶段

两段封锁协议(2PL: two-Phase Locking protocal)

- 读写数据之前要获得锁。每个事务中所有封锁请求先于任何一个解锁请求
- 两阶段：加锁段，解锁段。加锁段中不能有解锁操作，解锁段中不能有加锁操作

T1	T2
Lock-X(B)	Lock-S(B)
$R_1(B)$	$R_2(B)$
B := B-50;	Lock-S(A)
$W_1(B)$	$R_2(A)$
Lock-X(A)	Display(A+B)
Unlock(B)	Unlock(A)
$R_1(A)$	Unlock(B)
A := A+50;	
$W_1(A)$	
Unlock(A)	

产生调度并执行：T_1在T_2前开始

T1	T2
Lock-X(B);	Lock-S(B): 等待
$R_1(B)$;	
B := B-50;	
$W_1(B)$	
Lock-X(A)	
Unlock(B)	
$R_1(A)$;	Lock-S(B);
A := A+50;	$R_2(B)$
$W_1(A)$	
Unlock(A)	
	Lock-S(A)
	$R_2(A)$;
	Display(A+B)
	Unlock(A); Unlock(B)

两段封锁协议

两段封锁协议可能产生死锁?

两段锁协议是可能产生“死锁”的协议!

T1	T2
Lock-X(B)	Lock-S(A)
$R_1(B)$	Lock-S(B)
B := B-50;	$R_2(B)$
$W_1(B)$	$R_2(A)$
Lock-X(A)	Display(A+B)
Unlock(B)	Unlock(A)
$R_1(A)$	Unlock(B)
A := A+50;	
$W_1(A)$	
Unlock(A)	

T1	T2
Lock-X(B);	Lock-S(A)
$R_1(B)$;	
B := B-50;	
$W_1(B)$	
Lock-X(A): 等待	Lock-S(B): 等待
Unlock(B)	
	$R_2(B)$
$R_1(A)$;	$R_2(A)$;
A := A+50;	Display(A+B)
$W_1(A)$	Unlock(A);
Unlock(A)	Unlock(B)

基于时间戳的并发控制方法

基于时间戳的并发控制方法

什么是时间戳?

时间戳(TIMESTAMP)

- 一种基于时间的标志，将某一时刻转换成的一个数值。
- 时间戳具有唯一性和递增性。

事务的时间戳

- 事务T启动时，系统将该时刻赋予T，为T的时间戳
- 时间戳可以表征一系列事务执行的先后次序:

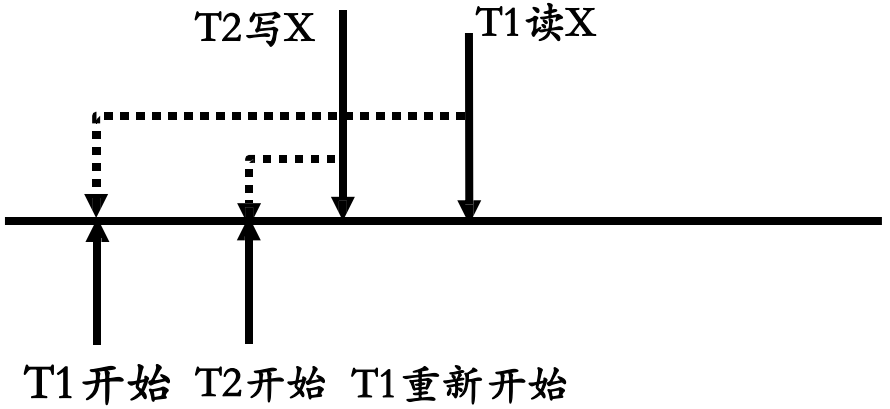
  时间戳小的事务先执行，时间戳大的事务后执行。
- 利用时间戳，可以不用锁，来进行并发控制

核心思想:

如果事务T_i的动作p和事务T_j的动作q冲突，且T_i比T_j时间戳早，则p必须在q之前执行。

否则，退出并重启违反该规则的事务。

冲突：读-写、写-读、写-写。



基于时间戳的并发控制方法

基于时间戳的简单调度规则

如何实现该调度规则？

对DB中的每个数据元素x，系统保留其上的最大事务时间戳

RT(x): 即R-timestamp(x)

读过该数据事务中最大的时间戳，即读过x的事务中最晚开始的事务的时间戳。

WT(x): 即W-timestamp(x)

写过该数据事务中最大的时间戳，即写过x的事务中最晚开始的事务的时间戳。

事务的时间戳

TS(T): 即TimeStamp

基于时间戳的并发控制方法

基于时间戳的简单调度规则

- **如何实现该调度规则?**

- 若T事务**读 read**(x)，则将T的时间戳TS(T)与WT(x)比较（解决读-写冲突）
  - 若TS(T)>WT(x) (T后开始)，则允许T操作，并且更改RT(x)为max{RT(x),TS(T)}；
  - 否则，有读写冲突，撤回T，重启T。

- 若T事务**写 write**(x)，(解决写-读和写-写冲突)
  - 若TS(T)<RT(x)，表示已有更晚开始的事务读了x。则**拒绝**本次操作撤回T重做。
  - 若TS(T)<WT(x)，表示已有更晚开始的事务已写过x， 则**拒绝**本次操作撤回T重做。
  - 其他情况，允许T写，并且更改WT(x)为T的时间戳：WT(x) = TS(T)；

基于时间戳的并发控制方法

基于时间戳的简单调度规则

示例

T1	T2	T3	A	B	C
200	150	175	RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
$r_1(B)$				RT=200	
	$r_2(A)$		RT=150		
		$r_3(C)$			RT=175
$w_1(B)$				WT=200	
$w_1(A)$			WT=200		
	$w_2(C)$				

读操作r: 若TS(T)>WT(x)

RT(x):= max{RT(x),TS(T)};

否则撤回、重做T

写操作w: 若

TS(T)≥max(WT(x), RT(x))

WT(x):= TS(T);

否则撤回、重做T

TS(T2) = 150 < max(0, 175), 冲突，T2撤销、重启！

基于时间戳的并发控制方法

基于时间戳的简单调度规则

示例

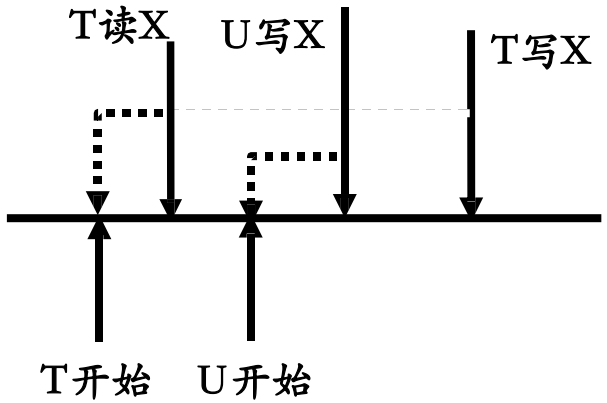
T1	T2	T3	A	B	C
200	205	175	RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
$r_1(B)$				RT=200	
		$r_3(C)$			RT=175
$w_1(B)$				WT=200	
$w_1(A)$			WT=200		
	$r_2(A)$				
	$W_2(C)$				

T2撤销、重启！

基于时间戳的另一种调度规则

需要解决的问题

如何放行一些事实上可实现的冲突?—托马斯写规则



修改之前关于写操作的判定规则：

若T事务写 write(x)，

- 若TS(T)<RT(x)，表示已有更晚开始的事务读了x。则**拒绝**本次操作撤回T重做。
- 若TS(T)<WT(x)，表示已有更晚开始的事务已写过x， 则**忽略本次写操作（不重启T）**
- 其他情况，允许T写，并且更改WT(x)为T的时间戳：WT(x) = TS(T)；

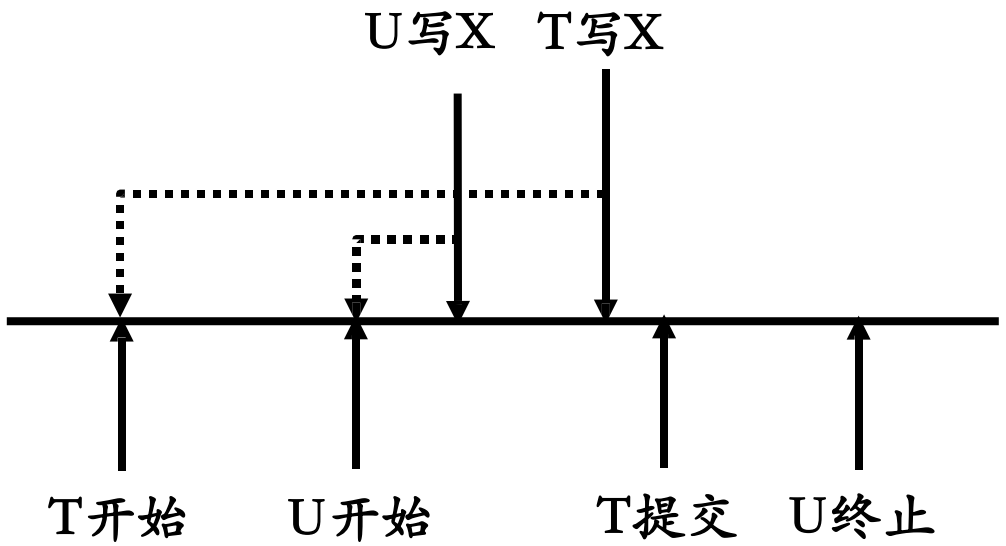
托马斯写规则

过时的写操作可直接被忽略，而无需撤销过时的事务

基于时间戳的另一种调度规则

需要解决的问题

托马斯规则也需注意的问题



T由于U的写入而被跳过(无需写)，但U却终止了

基于时间戳的另一种调度规则(提交位)

另一种调度规则

另一种调度规则

- 对DB中的每个数据元素x，系统保留其上的最大时间戳
  - RT(x): 即R-timestamp(x)
  - 读过该数据事务中最大的时间戳，即最后读x的事务的时间戳。
  - WT(x): 即W-timestamp(x)
  - 写过该数据事务中最大的时间戳，即最后写x的事务的时间戳。
  - C(x): x的提交位。
  - 该位为真，当且仅当最近写x的事务已经提交。
    - C(x)的目的是避免出现事务读另一事务U所写数据然后U终止这样的情况。

如何利用提交位?

事务的时间戳

TS(T): 即TimeStamp

基于时间戳的另一种调度规则

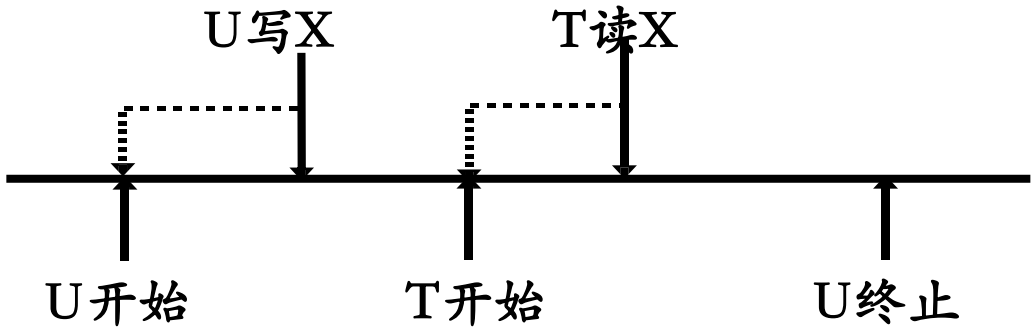
另一种调度规则

对来自事务T的读写请求，调度器可以：

同意请求

撤销/终止T，并重启具有新时间戳的T(终止+重启)

推迟T，并在以后决定是终止T还是同意请求(否则：如果请求是读，此读可能是脏的)



调度规则

假设调度器收到请求$r_T(X)$

(1)如果TS(T)>=WT(x), 此读是事实上可实现的

如C(x)为真，同意请求。如果TS(T)>RT(x), 置RT(x):=TS(T); 否则不改变RT(x).

如C(x)为假，推迟T直到C(x)为真或写x的事务终止。

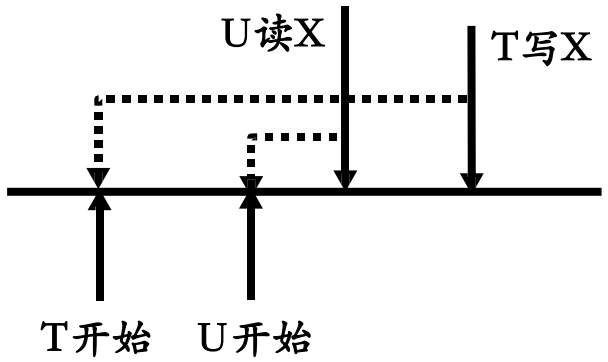
(2)如果TS(T)<WT(x), 此读是事实上不可实现的

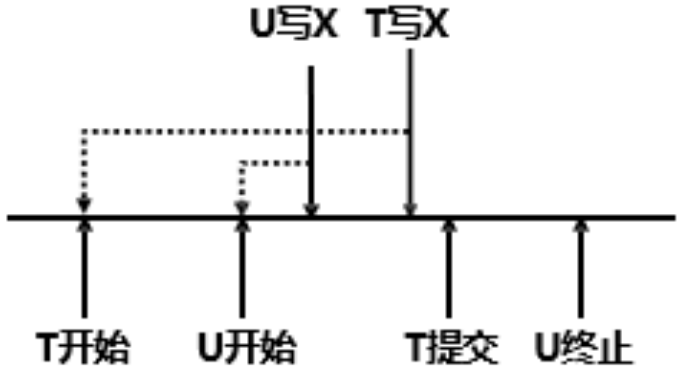
回滚T(终止并重启T)；(过晚的读)

基于时间戳的另一种调度规则

另一种调度规则

- 假设调度器收到请求$w_T(X)$
- (1)如果TS(T)>=RT(x), 且TS(T)>=WT(x), 此写是事实上是可实现的
- 为x写入新值； 置WT(x):=TS(T)； 置C(x):=false.
- (2)如果TS(T)>=RT(x),但是TS(T)<WT(x)， 此写是事实上可实现的。但x 已经有一个更晚的值
- 如果C(x)为真，那么前一个x的写已提交；则忽略T的写；继续进行。(托马斯写规则)
- 如果C(x)为假，则我们需推迟T，直到C(x)为真或写x的事务终止。
- (3)如果TS(T)<RT(x), 此写是事实上不可实现的
- T必须回滚。(过晚的写)

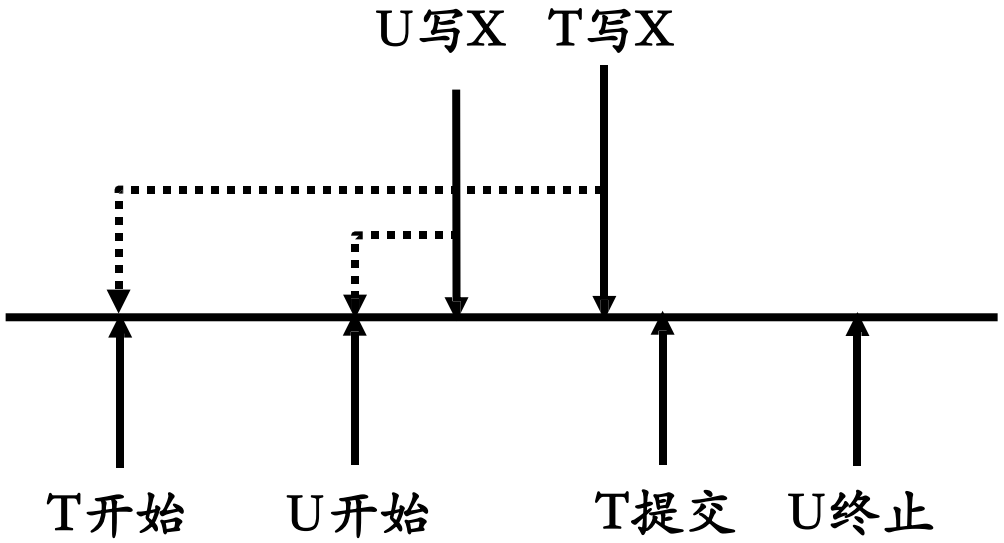




基于时间戳的另一种调度规则

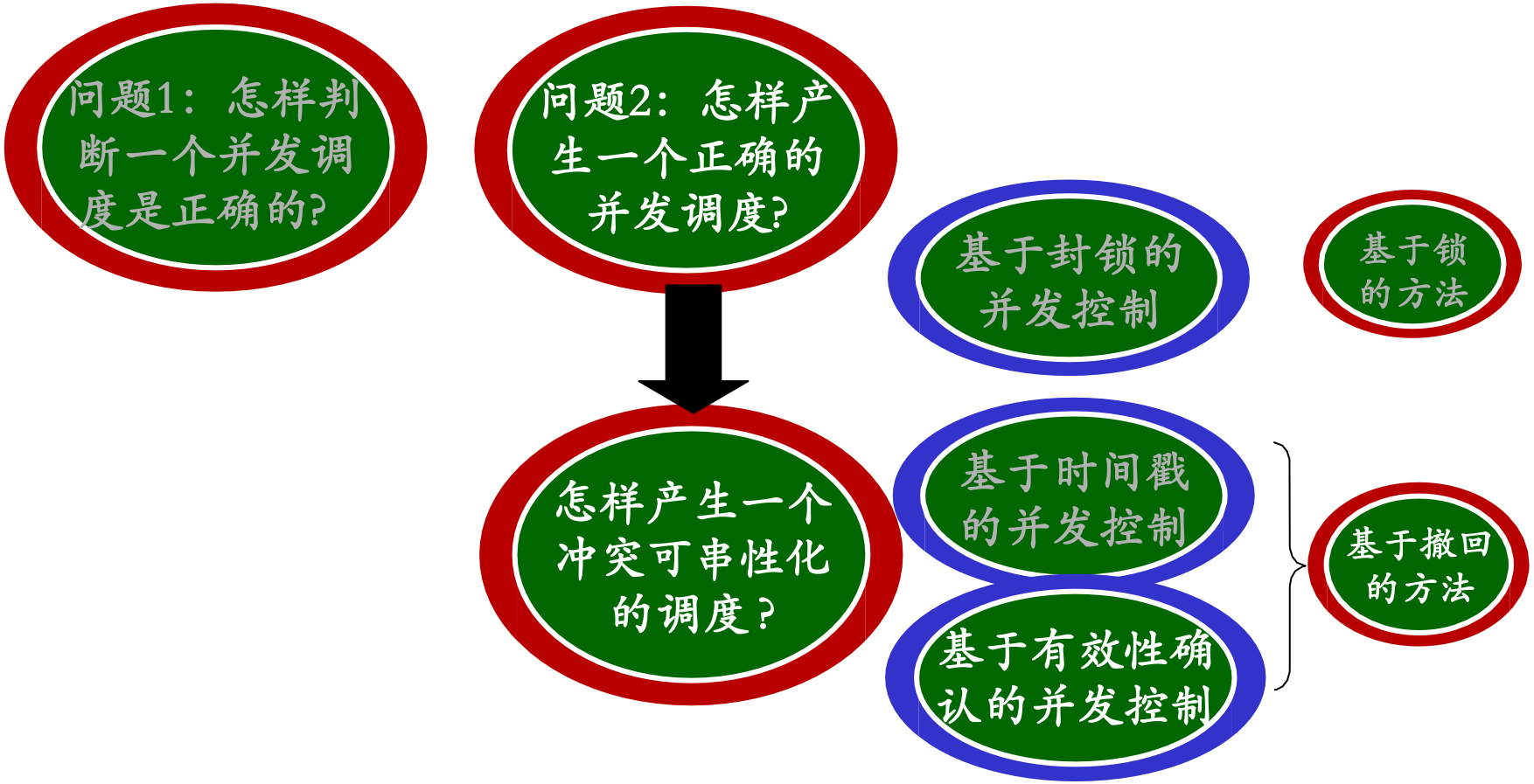
另一种调度规则

- 假设调度器收到提交T的请求。
- 它必须找到T所写的所有数据库元素x，并置C(x):=true。
- 如果有任何等待x被提交的事务，这些事务就被允许继续进行。
- 假设调度器收到终止T的请求
- 像前述步骤一样确定回滚T。那么任何等待T所写元素x的事务必须重新尝试读或写，看这一动作现在T的写被终止后是否合法。



基于有效性确认的并发控制方法（自学部分）

基于有效性确认的并发控制方法



能否进行批量性的冲突检测?

基于有效性确认的并发控制方法

什么是有效性确认?

- 基于时间戳的并发控制的思想
  - 事务在启动时刻被赋予唯一的时间戳，以示其启动顺序。
  - 为每一数据库元素保存读时间戳和写时间戳，以记录读或写该数据元素的最后的事务。
  - 通过在事务读写数据时判断是否存在冲突(读写冲突、写读冲突、写写冲突) 来强制事务以可串行化的方式执行。
- 基于有效性确认的并发控制的思想
  - 事务在启动时刻被赋予唯一的时间戳，以示其启动顺序。
  - 为每一活跃事务保存其读写数据的集合。RS(T)：事务T读数据的集合； WS(T)：事务T写数据的集合。
  - 通过对多个事务的读写集合，判断是否有冲突(存在事实上不可实现的行为)，即有效性确认，来完成事务的提交与回滚，强制事务以可串行化的方式执行。

怎么落实呢?

基于有效性确认的并发控制方法

基于有效性确认的调度器?

基于有效性确认的调度器

事务在启动时刻被赋予唯一的时间戳，以示其启动顺序。

每一事务读写数据的集合。RS(T)：事务T读数据的集合；WS(T)：事务T 写数据的集合。

事务分三个阶段进行

读阶段。事务从数据库中读取读集合中的所有元素。事务还在其局部地址空间计算它将要写的所有值;

有效性确认阶段。调度器通过比较该事务与其它事务的读写集合来确认该事务的有效性。

写阶段。事务往数据库中写入其写集合中元素的值。

每个成功确认的事务是在其有效性确认的瞬间执行的。

并发事务串行的顺序即事务有效性确认的顺序。

基于有效性确认的并发控制方法

基于有效性确认的调度器?

- 调度器维护三个集合
  - START集合。已经开始但尚未完成有效性确认的事务集合。对此集合中的事务，调度器维护START(T)，即事务T开始的时间。
  - VAL集合。已经确认有效性但尚未完成第3阶段写的事务。对此集合中的事务，调度器维护START(T)和VAL(T)，即T确认的时间。
  - FIN集合。已经完成第3阶段的事务。对这样的事务T，调度器记录START(T), VAL(T)和FIN(T)，即T完成的时间。

基于有效性确认的并发控制方法

有效性确认规则?

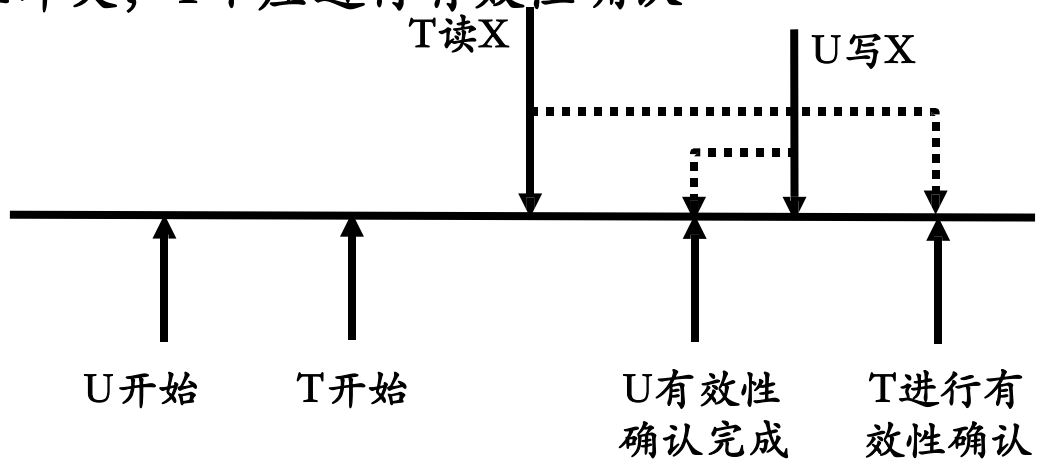
冲突一：假设存在事务U 和 T满足:

(1)U 在VAL或FIN中, 即U已经过有效性确认。

(2)FIN(U)>START(T), 即U在T开始前没有完成。

(3)RS(T) ∩ WS(U)非空, 特别地，设其均包含数据库元素为x。

则T和U的执行存在冲突，T不应进行有效性确认



如果一个较早的事务U现在正在写入T应该读过的某些对象，则T的有效性不能确认

基于有效性确认的并发控制方法

基于有效性确认的并发控制方法

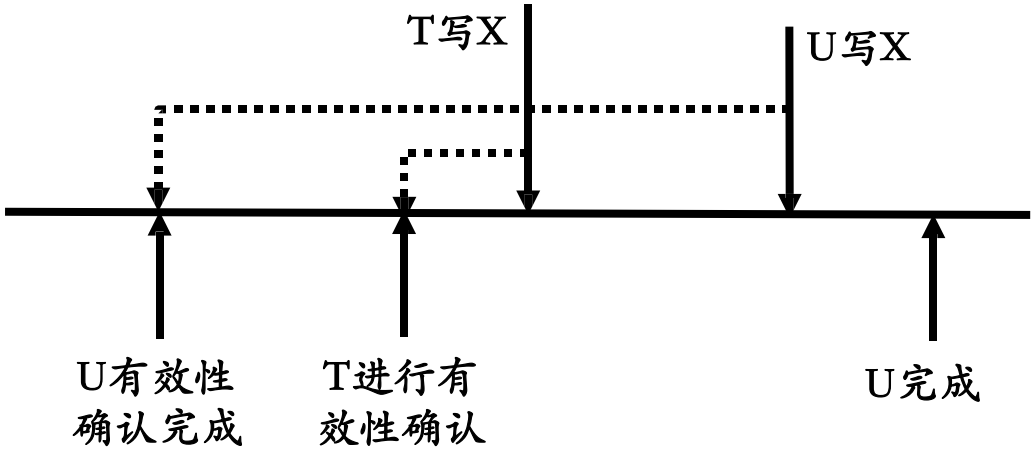
冲突二：假设存在事务U 和T 满足：

(1)U 在VAL, 即U有效性已经成功确认。

(2)FIN(U)>VAL(T), 即U在T进入其有效性确认阶段以前没有完成。

(3)WS(T) ∩ WS(U)非空, 特别地，设其均包含数据库元素x。

则T和U的执行存在冲突，T不应进行有效性确认



如果T在有效性确认后可能比一个较早的事务先写某个对象，则T的有效性不能确认

基于有效性确认的并发控制方法

有效性确认规则?

有效性确认规则

(1)对于所有已经过有效性确认, 且在T开始前没有完成的U, 即对于满足FIN(U)>START(T)的U,检测:

RS(T)　WS(U)是否为空。

若为空，则确认。否则，不予确认。

(2)对于所有已经过有效性确认，且在T有效性确认前没有完成的U, 即对于满足FIN(U)>VAL(T)的U, 检测:

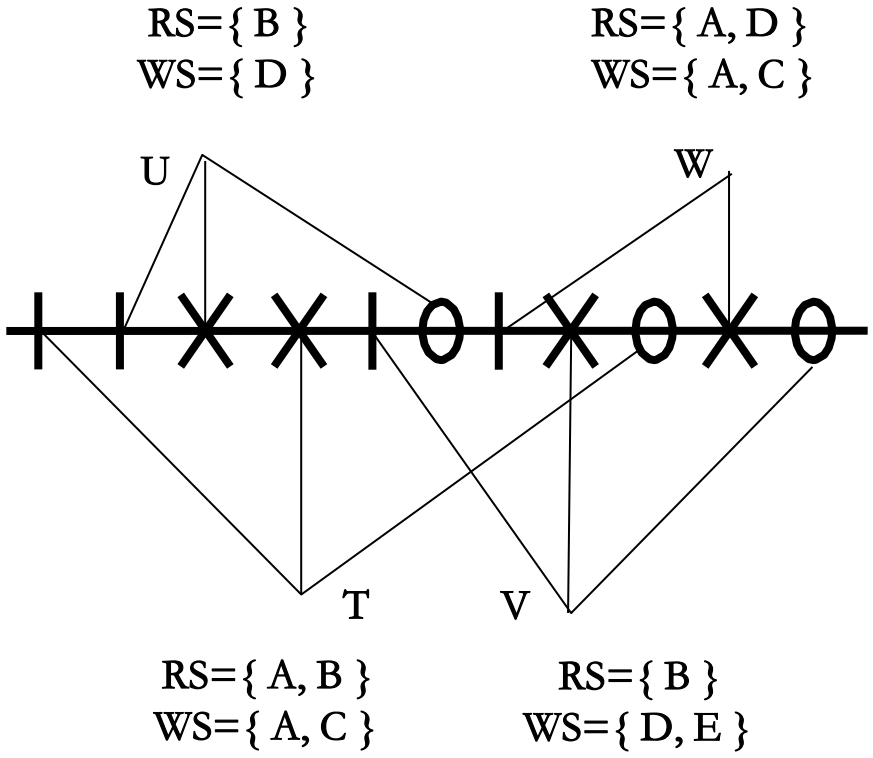
WS(T)　WS(U)是否为空。

若为空，则确认。否则，不予确认。

基于有效性确认的并发控制方法

有效性确认规则?

示例：确认下列四个事务的有效性



| =开始； X =有效性确认； O =完成

1. U的有效性确认

无需检测，直接确认U。

2. T的有效性确认

因FIN(U)>START(T), 需检测RS(T)　WS(U) 因

FIN(U)>VAL(T),需检测WS(T)　WS(U)

检测结果：均为空，则确认T。

3. V的有效性确认

因FIN(U)>START(V), 需检测RS(V)　WS(U) 因

FIN(T)>START(V),需检测RS(V)　WS(T) 因

FIN(T)>VAL(V),需检测WS(T)　WS(V)

检测结果：均为空，则确认V。

4. W的有效性确认

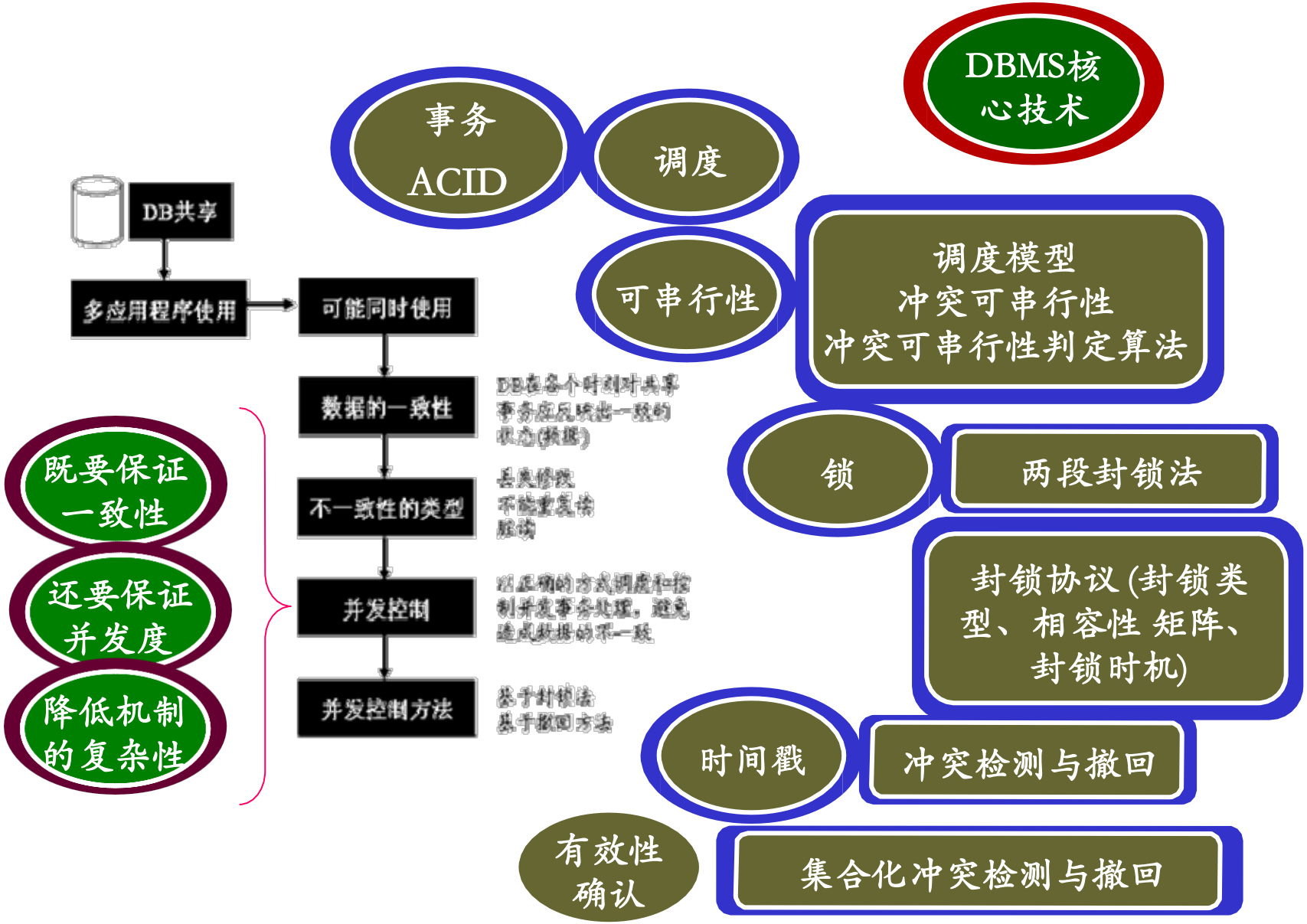
因FIN(T)>START(W)，需检测RS(W)　WS(T)　因

FIN(V)>START(W),需检测RS(W)　WS(V)　因

FIN(V)>VAL(W),需检测WS(V)　WS(W)

检测结果：不全为空,则W不能确认,W被回滚。

总结（Summary）


DBMS核心技术
事务 ACID
调度
可串行性
调度模型
冲突可串行性
冲突可串行性判定算法
锁
两段封锁法
封锁协议(封锁类型、相容性 矩阵、封锁时机)
时间戳
冲突检测与撤回
有效性确认
集合化冲突检测与撤回
DB共享
多应用程序使用
可能同时使用
数据的一致性
不一致性的类型
并发控制
并发控制方法
既要保证一致性
还要保证并发度
降低机制的复杂性