

数据库管理系统实现技术

数据库事务处理技术

(并发控制)

本讲学习什么？

基本内容

1. 为什么需要并发控制
2. 事务调度及可串行性
3. 基于封锁的并发控制方法
4. 基于时间戳的并发控制方法
5. 基于有效性确认的并发控制方法？

重点与难点

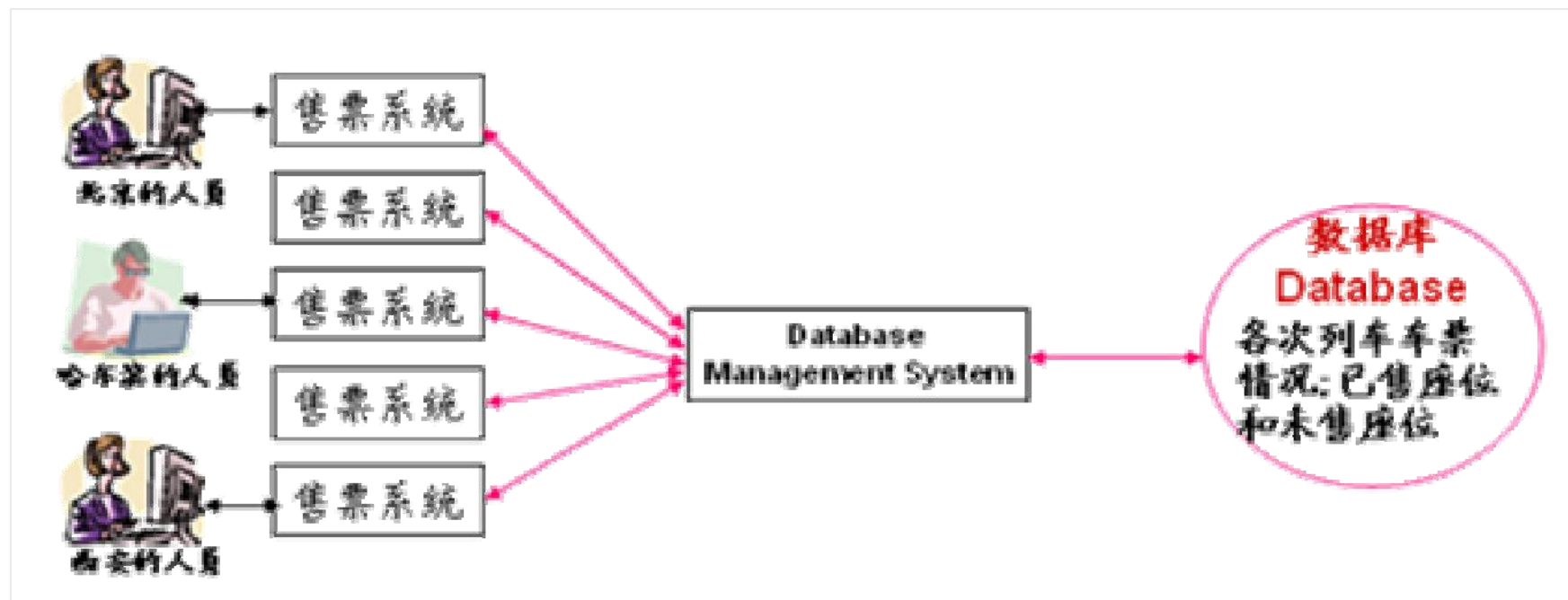
- 理解数据库并发操作的三种不一致性及其产生原因
- 理解一组概念：事务、事务调度、可串行性、时间戳等
- 掌握三种类型的并发控制方法：基于封锁的方法、基于时间戳的方法、基于有效性确认的方法
- 重点掌握：冲突可串行性判别算法，两段封锁法，基于时间戳的方法；

为什么要进行并发控制

为什么要进行并发控制

(1)数据库可能存在不一致

如果大家同时买起点终点、日期、车次相同的车票，会否买到座位相重复的车票？



为什么要进行并发控制

(2)三种典型的不一致现象

要理解不一致性是怎样发生的？

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-1 M: A=49)
Write A (M: A=49 DB : A=49)	
	Write A (M: A=49 DB : A=49)

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)	
	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)	

A1 ≠ A2 两次读的不是同一数据。

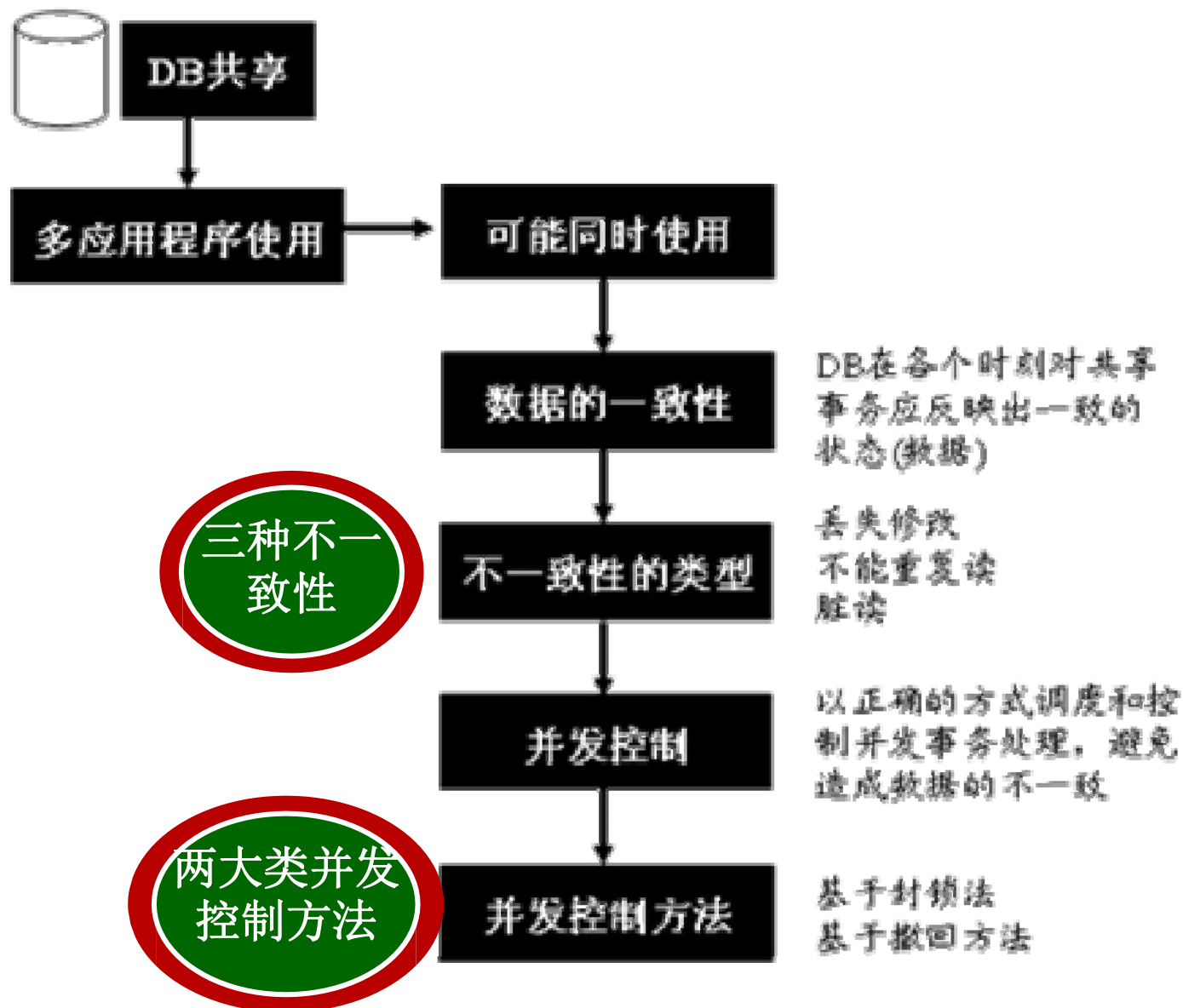
3. 脏读

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)	

A2 已无效，应为 A1。

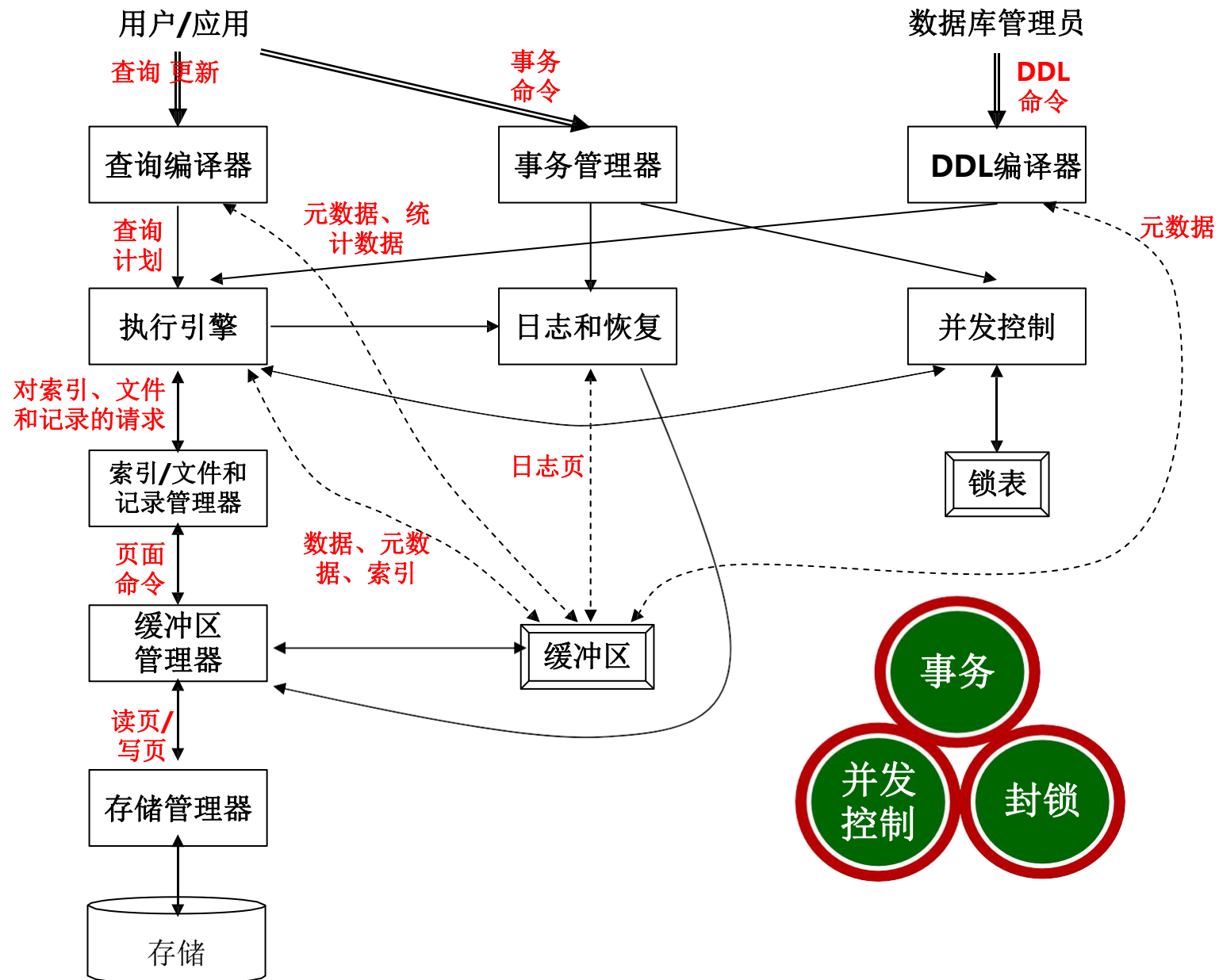
为什么要进行并发控制

(3)并发控制的缘由



为什么要进行并发控制

(4)并发控制及相应的事务处理技术是DBMS的核心技术



什么是事务

什么是事务

(1)事务的概念

[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5000RMB到帐户B上 T：

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

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。

什么是事务

(2)事务的宏观性和微观性

事务的宏观性(应用程序员看到的事务): 一个存取或改变数据库内容的程序的一次执行, 或者说一条或多条**SQL**语句的一次执行被看作一个事务。

➤ 事务一般是由应用程序员提出, 因此有开始和结束, 结束前需要提交或撤消。

Begin Transaction

exec sql ...

...

exec sql ...

exec sql commit work | exec sql rollback work End

Transaction

➤ 在嵌入式**SQL**程序中, 任何一条数据库操纵语句(如**exec sql select**等)都会引发一个新事务的开始, 只要该程序当前没有正在处理的事务。而事务的结束是需要应用程序员通过**commit**或**rollback**确认的。因此**Begin Transaction** 和**End Transaction**两行语句是不需要的。

什么是事务

(2)事务的宏观性和微观性

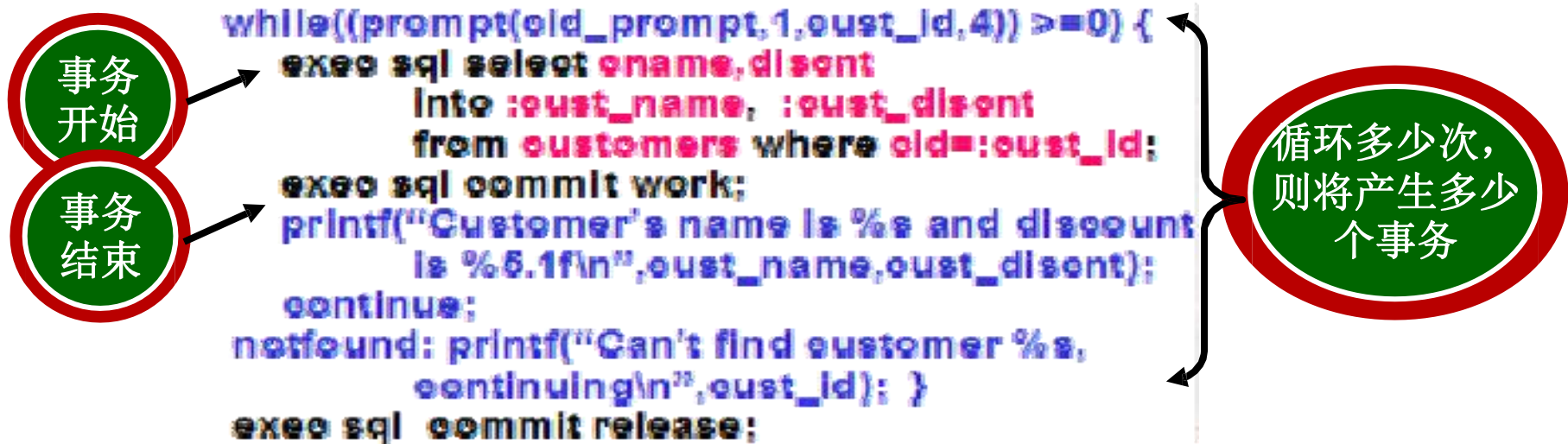
- 一个事务可处理一个数据或一条记录，如下例：

"Update T1 Set val = :pgmval2 where uniqueid = B; "

- 复杂一些的事务也可能处理一批数据或一批记录，如下例：

"Update T1 Set val = 1.15 * val where uniqueid between :low and :high; "

- 一段程序语句，可能会循环执行。执行中，由**SQL**语句引出事务，至**Commit/RollBack**结束事务，每次重复执行都将产生一个事务。



- 更为复杂的事务由多条**SQL**语句构成。

什么是事务

(2)事务的宏观性和微观性

事务的微观性(DBMS看到的事务): 对数据库的一系列基本操作(读、写)的一个整体性执行。

T: **read(A);**
 A := A - 5000;
 write(A);
 read(B);
 B := B + 5000;
 write(B);

事务的并发执行: 多个事务从宏观上看是并行执行的, 但其微观上的基本操作(读、写)则可以是交叉执行的。

什么是事务

(3)事务的特性

事务

宏观独立完整

微观交错执行

并发控制就是通过事务微观交错执行次序的正确安排,保证事务宏观的独立性、完整性和正确性

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-1 M: A=49)
Write A (M: A=49 DB : A=49)	
	WriteA(A ₂) (M: A=49 DB : A=49)

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)	
	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)	

A₁ ≠ A₂ 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)	

为 A₁。

三种不正确的次序安排则引发了不一致性

什么是事务

(3)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

□**一致性Consistency**: DBMS保证事务的操作状态是正确的, 符合一致性的操作规则, 不能出现三种典型的不一致性。它是进一步由隔离性来保证的。

□**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;或者先执行了T2, 再执行T1。

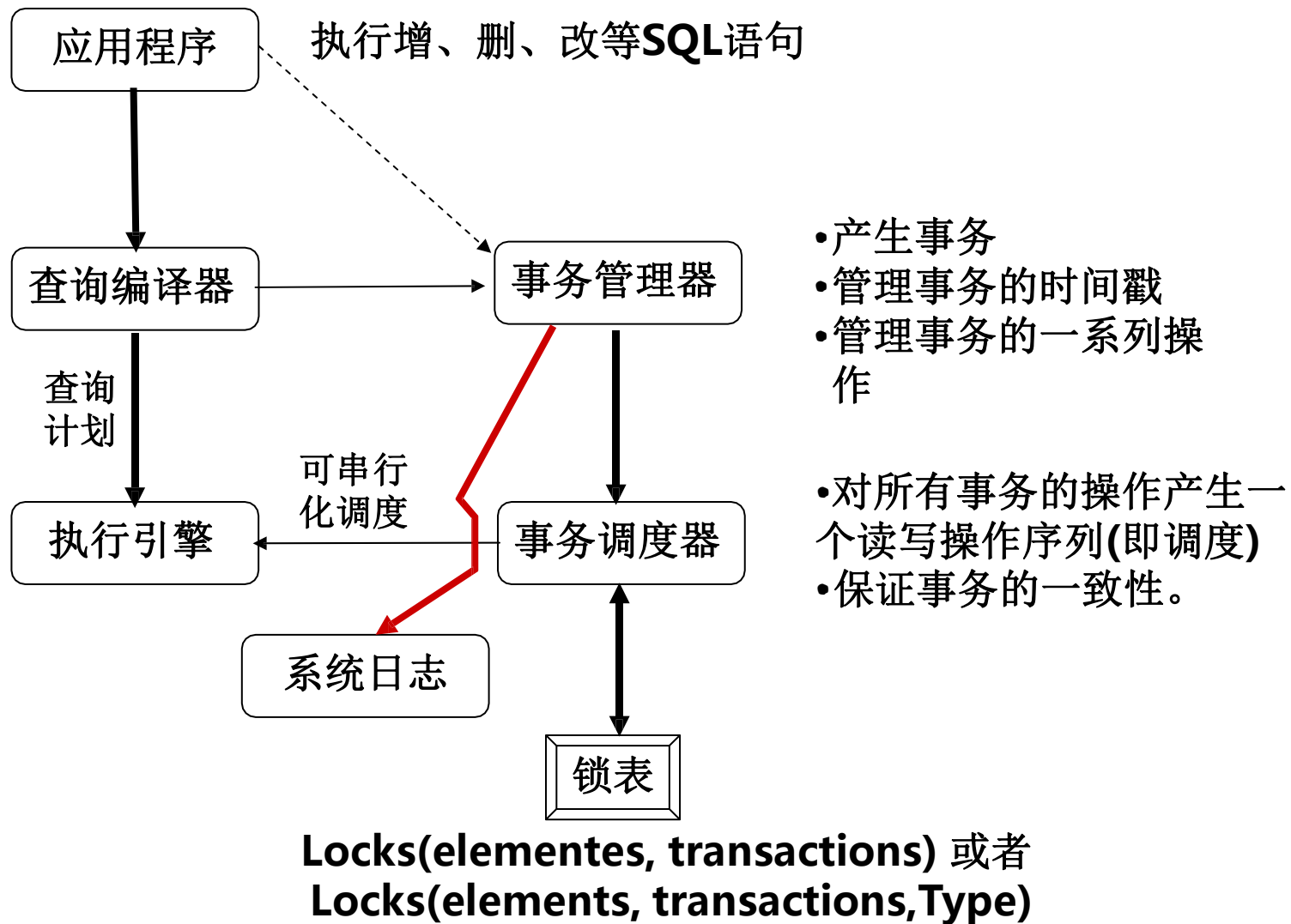
□**持久性Durability**: DBMS保证已提交事务的影响是持久的, 被撤销事务的影响是 可恢复的。

➤换句话说: 具有ACID特性的若干数据库基本操作的组合体被称为事务。

1. 丢失修改		2. 不能重复读		3. 脏读	
T1	T2	T1	T2	T1	T2
Read A (DB : A = 50 M: A = 50)		Read A (DB : A = A1 M: A = A1)		Read A (DB : A = A1 M: A = A1)	Read A (DB : A = A1 M: A = A1)
	Read A (DB : A = 50 M: A = 50)		Read A (DB : A = A1 M: A = A1)		Update A (M: A = A2)
Update A (设 A = A-1 M: A = 49)			Update A (M: A = A2)		Write A (M: A = A2 DB : A = A2)
	Update A (设 A = A-1 M: A = 49)		Write A (M: A = A2 DB : A = A2)	Read A (DB : A = A2 M: A = A2)	
Write A (M: A = 49 DB : A = 49)		Read A (DB : A = A2 M: A = A2)			Roll Back (DB : A = A1)
	WriteA(A ₂) (M: A = 49 DB : A = 49)			Read A (M: A = A2)	
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。		A ₁ ≠ A ₂ 两次读的不是同一数据。		A ₂ 已无效, 应为 A ₁ 。	

什么是事务

(4)DBMS对事务的控制



事务调度与可串行性

事务调度与可串行性

(1)基本概念

[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: 多个事务从宏观上看是并行执行的,但其微观上的基本操作(读、写)则是交叉执行的。

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

并发调度

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	
11		C=C+20
12		Write C

事务调度与可串行性

(1)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。

问题1：怎样判断一个并发调度是正确的？

问题2：怎样产生一个正确的并发调度？

[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

(1)基本概念

串行调度

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

可串行化调度

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

不可串行化调度

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	
11		C=C+20
12		Write C

- 可串行化调度一定是正确的并行调度，但正确的并行调度，却未必都是可串行化的调度。为什么？。
- 并行调度的正确性是指内容上结果正确性，而可串行性是指形式上结果正确性，便于操作(如右侧图T2中的B=B-20改为B=B-0,则调度是正确的，但是不可串行化)
- 可串行化的等效串行序列不一定唯一。

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

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

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

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事务的行为会改变。

➤有冲突的两个操作是不能交换次序的，没有冲突的两个事务是可交换的

➤几种冲突的情况：

✓同一事务的任何两个操作都是冲突的

$r_i(X); w_i(Y)$ $w_i(X); r_i(Y)$

✓不同事务对同一元素的两个写操作是冲突的

$w_i(X); w_j(X)$

✓不同事务对同一元素的一读一写操作是冲突的

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

冲突可串行性： 一个调度，如果通过交换相邻两个无冲突的操作能够 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。

并发调度
 $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

冲突的可串行化调度

交换操作的次序

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

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

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

得到了串行的调度

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

事务调度与可串行性

(3)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。

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

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

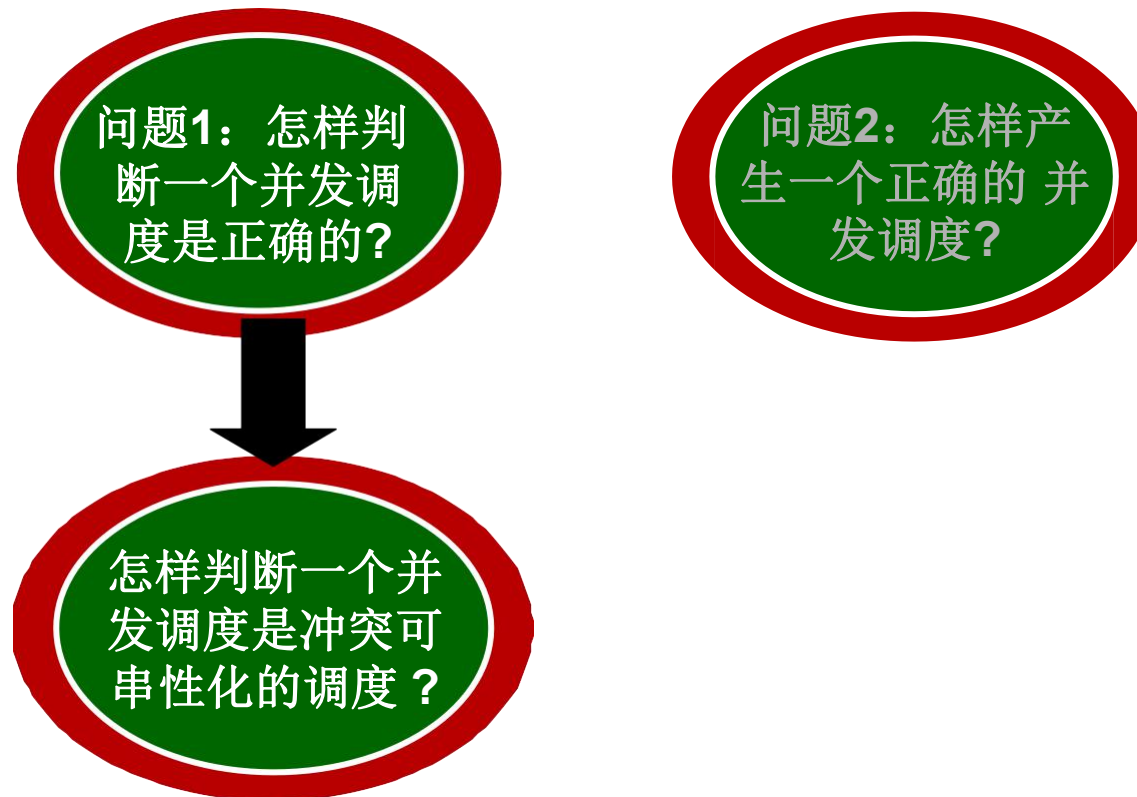
等效，但
不能无冲
突转换

冲突可串行性判别算法

冲突可串行性判别算法

(1)问题

- 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。
- 并发调度的正确性 \supseteq 可串行性 \supseteq 冲突可串行性



冲突可串行性判别算法

(2)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!

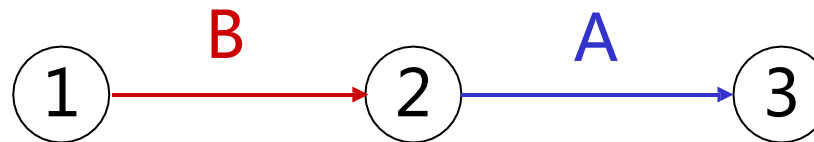
冲突可串行性判别算法

(3)示例

示例

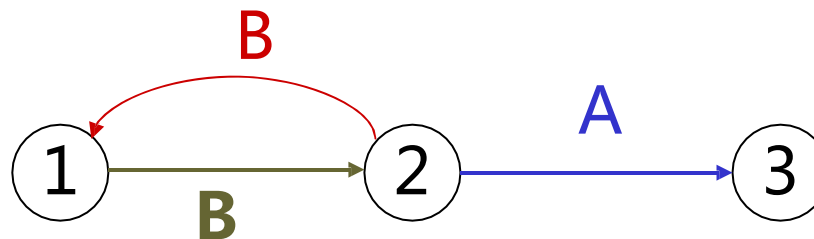
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

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

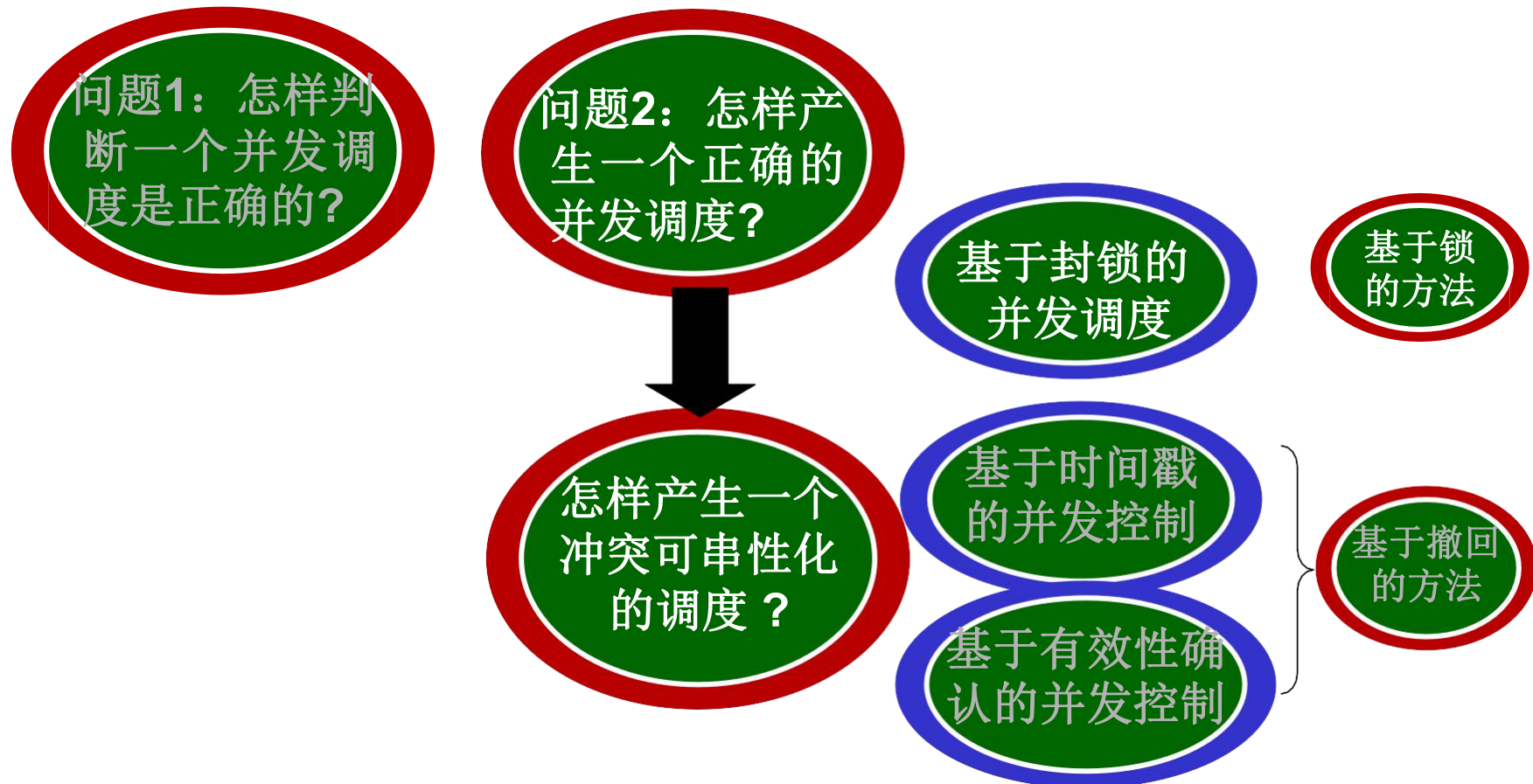


基于封锁的并发控制方法

基于封锁的并发控制方法

(1)问题

- 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。
- 并发调度的正确性 \supseteq 可串行性 \supseteq 冲突可串行性



基于封锁的并发控制方法

(2)什么是锁？

“锁” 是控制并发的一种手段

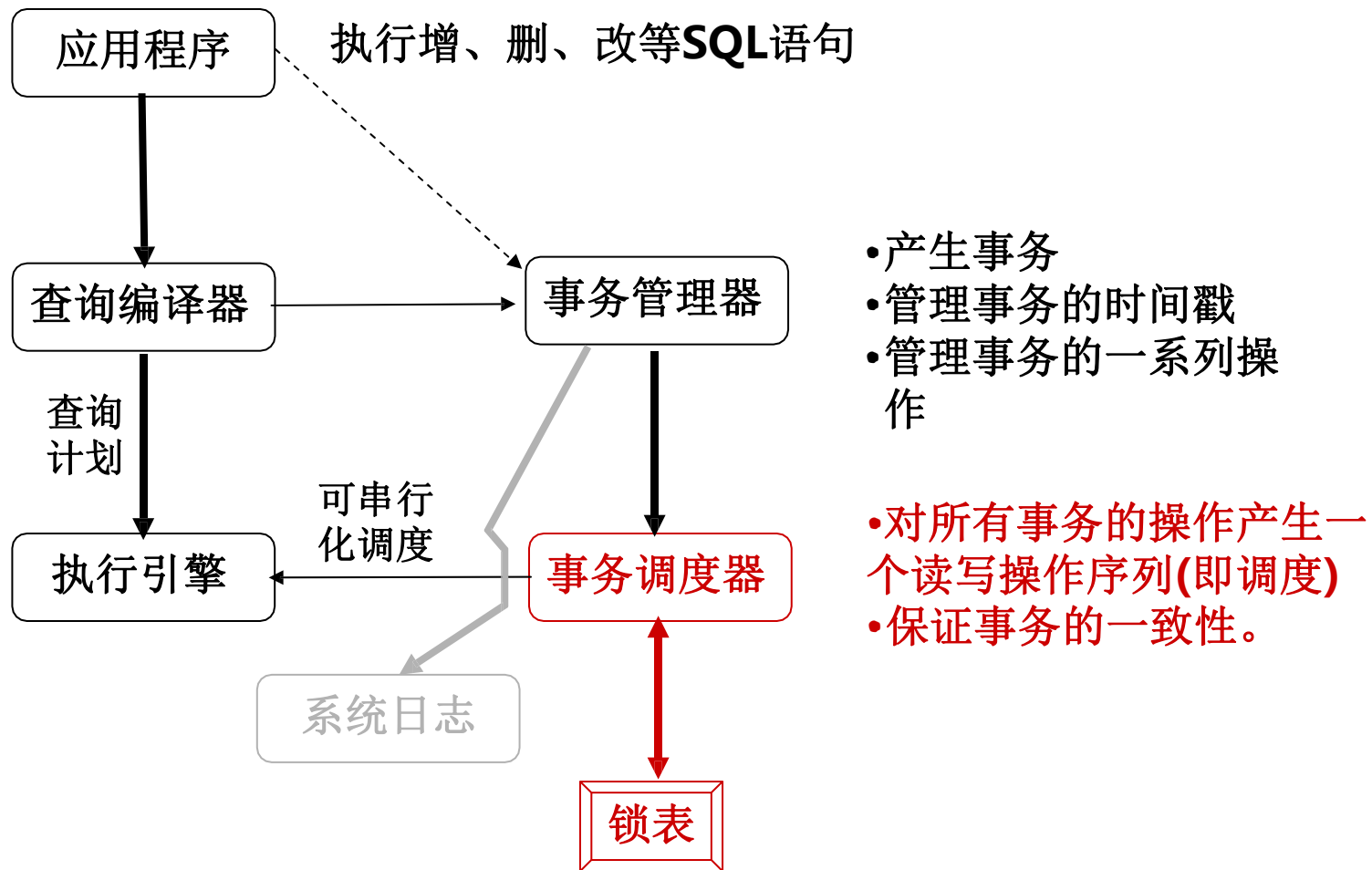
- 每一数据元素都有一唯一的锁
- 每一事务读写数据元素前，要获得锁。
- 如果被其他事务持有该元素的锁，则要等待。
- 事务处理完成后要释放锁。

$L_i(A)$: 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

(2)什么是锁？



**Locks(elementes, transactions) 或者
Locks(elements, transactions,Type)**

基于封锁的并发控制方法

(2)什么是锁？

- 调度器可利用锁来保证冲突可串行性

T1	T2
L₁(A); READ(A, t) t := t+100	
WRITE(A, t);	
U₁(A); L₁(B)	
	L₂(A); READ(A,s)
	s := s*2
	WRITE(A,s); U₂(A);
	L₂(B); “拒绝...”
READ(B, t) t := t+100 WRITE(B,t);	
U₁(B);	
	...获得 “锁”
	READ(B,s)
	s := s*2
	WRITE(B,s); U₂(B);

基于封锁的并发控制方法

(2)什么是锁？

- 锁本身并不能保证冲突可串行性。
- 锁为调度提供了控制的手段。但如何用锁，仍需说明。---不同的协议

T1	T2
L₁(A); READ(A, t)	
t := t+100	
WRITE(A, t); U₁(A);	
	L₂(A); READ(A,s)
	s := s*2
	WRITE(A,s); U₂(A);
	L₂(B); READ(B,s)
	s := s*2
	WRITE(B,s); U₂(B);
L₁(B); READ(B, t)	
t := t+100	
WRITE(B,t); U₁(B);	

“锁”

调度

冲突可串行性

封锁协议之锁的类型

- 排他锁**X** (e**X**clusive locks)

只有一个事务能读、写，其他任何事务都不能读、写

- 共享锁**S** (**S**hared locks)

所有事务都可以读，但任何事务都不能写


- 更新锁**U** (**U**ppdate locks)

初始读，以后可升级为写

- 增量锁**I** (**I**ncremental lock)

增量更新(例如 $A=A+x$)

区分增量更新和其他类型的更新



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

(3)封锁协议需要考虑什么？

封锁协议之相容性矩阵

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

如何表达封锁协议？

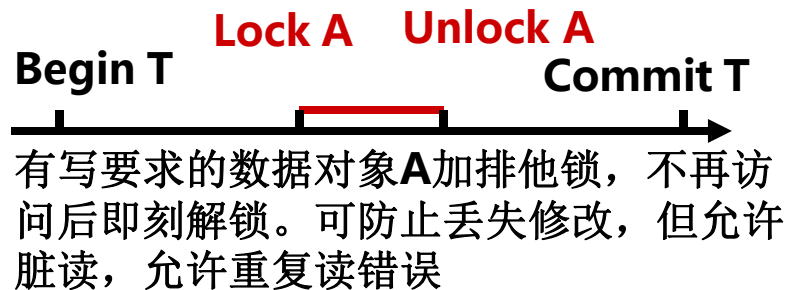
这只是简单形式，还有更丰富内容

基于封锁的并发控制方法

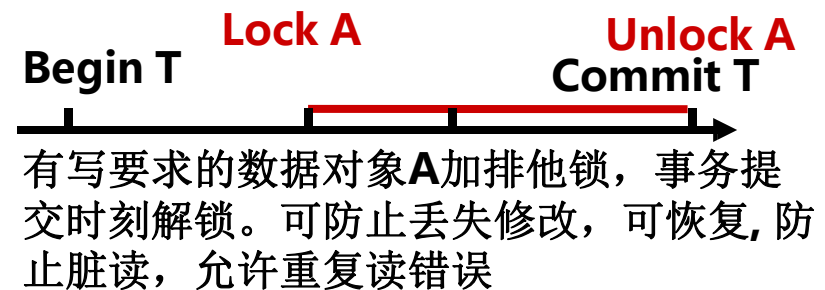
(3)封锁协议需要考虑什么？

封锁协议之加锁/解锁时机

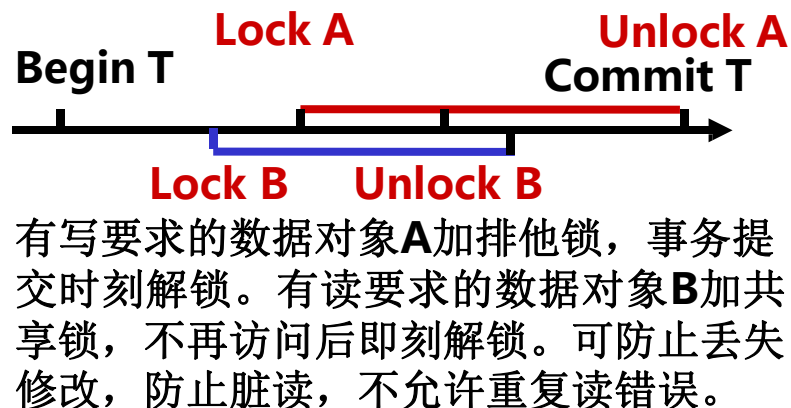
0级协议(0-LP)



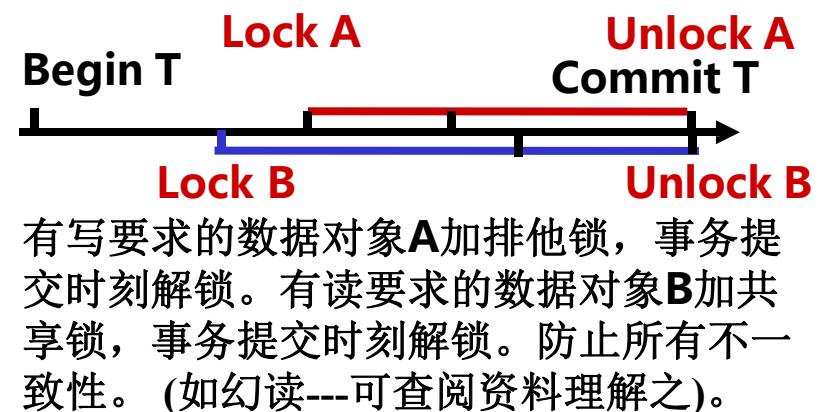
1级协议(1-LP)



2级协议(2-LP)



3级协议(3-LP)



基于封锁的并发控制方法

(3)封锁协议需要考虑什么？

SQL之隔离性级别(允许程序员选择使用)

读未提交 (read uncommitted) ---相当于0级协议

读已提交 (read committed) ---相当于1级协议

可重复读(repeatable read) ---相当于2级协议

可串行化(serializable) ---相当于3级协议

	脏读	重复读错误	幻读
读未提交	允许	允许	允许
读已提交	不允许	允许	允许
可重复读	不允许	不允许	允许
可串行化	不允许	不允许	不允许

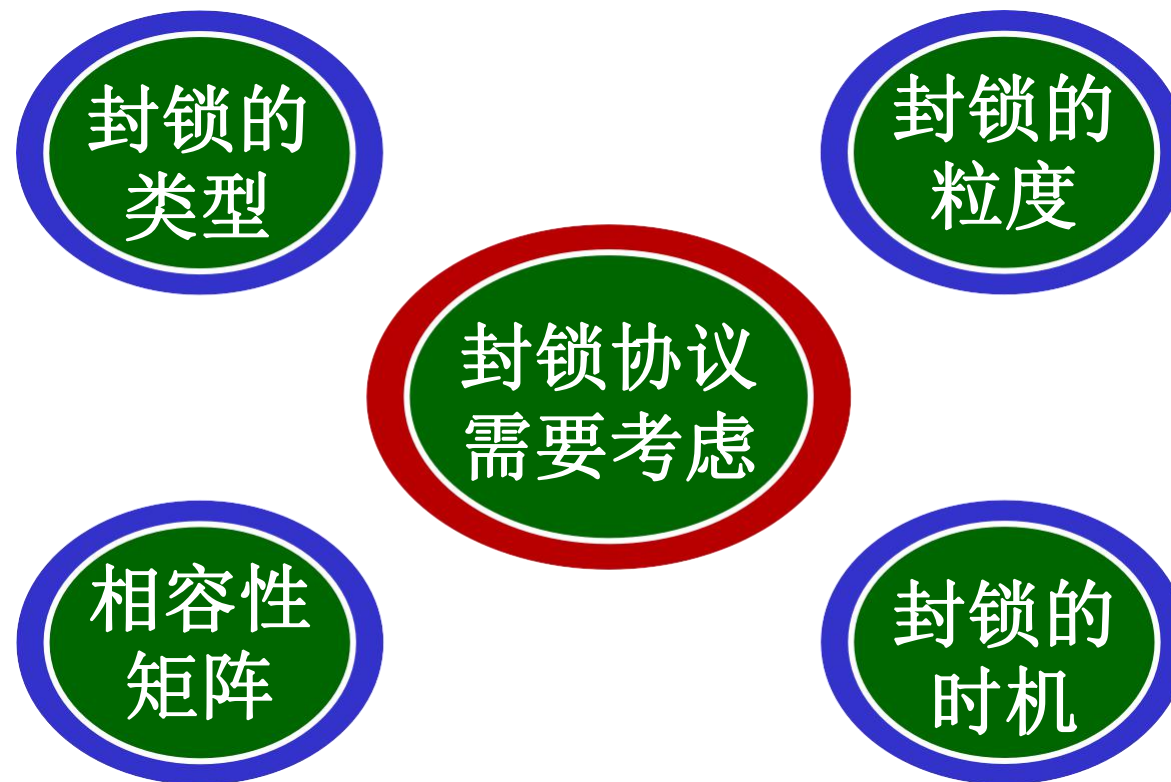
幻读指的是事务不是串行发生时的一种现象，是事务A读取了事务B已提交的新增数据。例如第一个事务对一个表的所有数据进行修改，同时第二个事务向表中插入一条新数据。那么操作第一个事务的用户就发现表中还有没有修改的数据行，就像发生了幻觉一样。解决幻读的方法是增加范围锁（range lock）或者表锁。

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。

基于封锁的并发控制方法

(3)封锁协议需要考虑什么？



衍生出不同的封锁协议

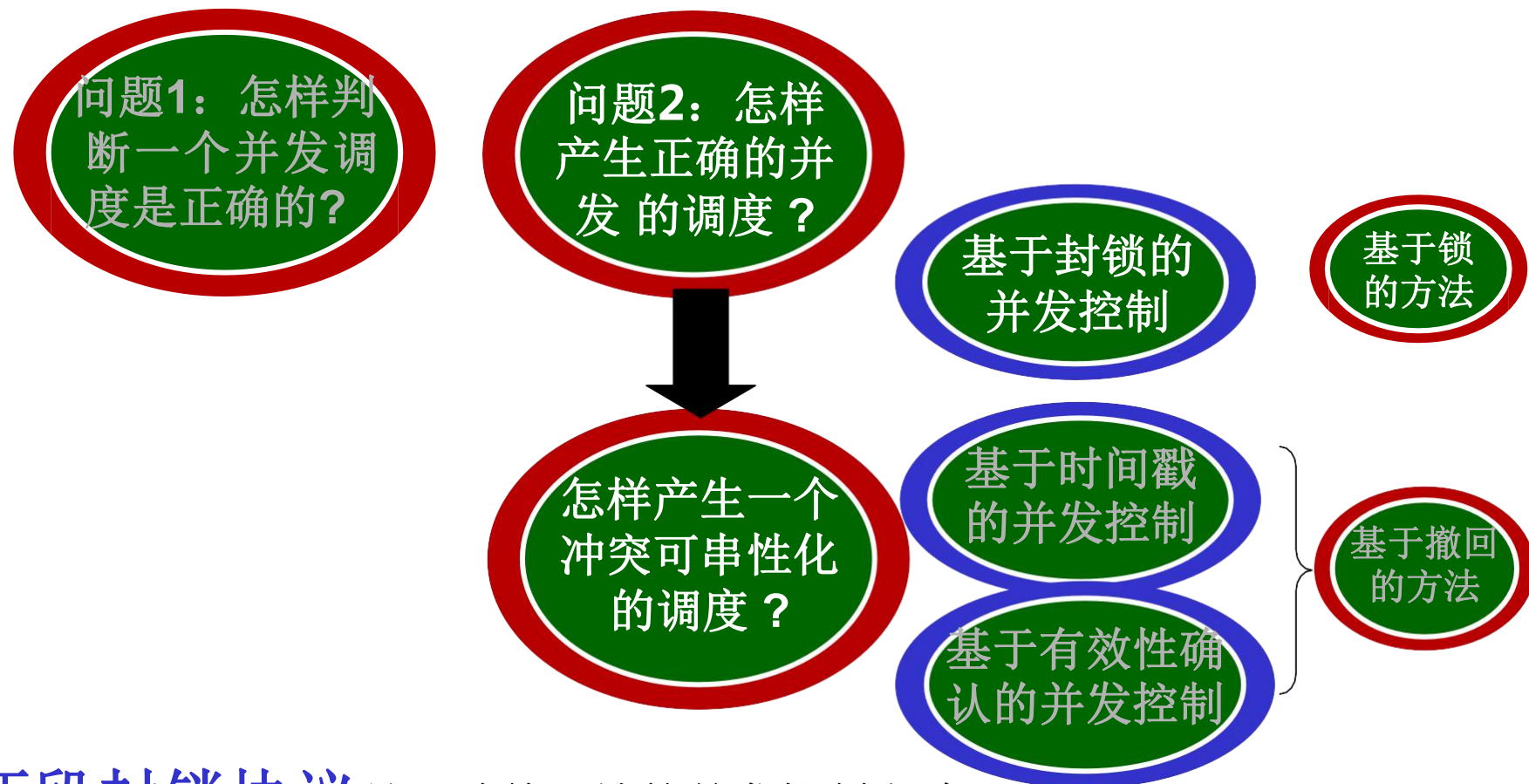
两段封锁协议

两段封锁协议

(0)问题

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。

●并发调度的正确性 \supseteq 可串行性 \supseteq 冲突可串行性



两段封锁协议是一种基于锁的并发控制方法

两段封锁协议

(1)什么是两段封锁协议?

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

T1	T2
READ(A, t)	READ(A,s)
t := t+100	s := s*2
WRITE(A, t);	WRITE(A,s);
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t);	WRITE(B,s);

按两段锁协议加锁解锁

T₁事务在U₁(A)后再不能对任何对象有加锁操作

T1	T2
L ₁ (A);	L ₂ (A);
L ₁ (B);	READ(A,s)
READ(A, t)	s := s*2
t := t+100	WRITE(A,s);
WRITE(A, t);	L ₂ (B);
U ₁ (A)	READ(B,s)
READ(B, t)	s := s*2
t := t+100	WRITE(B,s);
WRITE(B,t);	U ₂ (A);
U ₁ (B);	U ₂ (B);

加锁操作

解锁操作

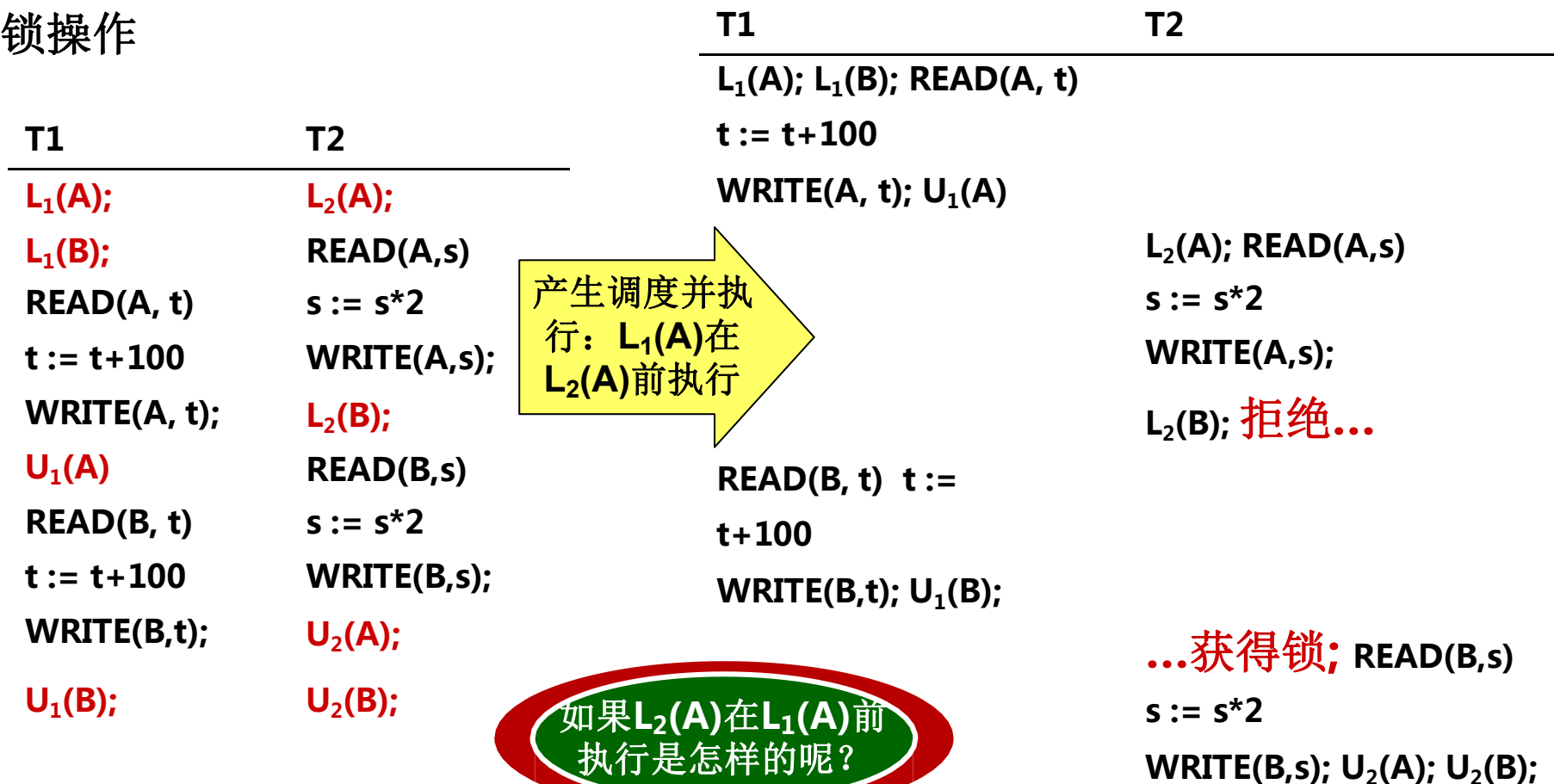
每个事务

两段封锁协议

(1)什么是两段封锁协议?

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



两段封锁协议

(2)两段封锁协议一定是冲突可串行化的？

两段封锁协议是可以保证冲突可串行性的！

归纳法证明：n-1个事务的2PL是可串行化的，那n个事务的2PL是否也是？

➤基础：n=1, S是一个可串行化调度。

➤归纳：假设S涉及n个事务 T_1, \dots, T_n ，并设 T_i 是在整个S中有第一个解锁动作如 $U_i(X)$ 的事务。按归纳再假设除 T_i 外的由其他n-1个事务构成的调度是一个满足可串行性的2PL调度。能否推断出：将 T_i 的所有动作进行无冲突的交换操作向前移动到调度的开始是可能的。

➤考虑 T_i 的某个动作，例如 $w_i(Y)$ 。S中这一动作前可能有冲突的动作吗，例如 $w_j(Y)$ 吗？如果有，那么在调度S中， $U_j(Y)$ 和 $L_i(Y)$ 必然交错出现在这样一个序列中：

$\dots w_j(Y); \dots U_j(Y); \dots; L_i(Y); \dots; w_i(Y); \dots$

➤既然 T_i 是第一个解锁的，S中 $U_i(X)$ 必然在 $U_j(Y)$ 前；也就是说，S可能形如

$\dots; w_j(Y); \dots; U_i(X); \dots; U_j(Y); \dots; L_i(Y); \dots; w_i(Y); \dots$

或 $U_i(X)$ 甚至可能出现在 $w_j(Y)$ 前。不管哪种情况， $U_i(X)$ 出现在 $L_i(Y)$ 前，与两段封锁协议矛盾。

➤所以： $w_i(Y)$ 前不可能有冲突的动作(对其他冲突动作可一样证明)。

➤结论是：确实能够通过先使用无冲突读写动作的交换，然后恢复 T_i 的加锁和解锁动作，将 T_i 的所有动作前移到S的开始。

(T_i 的动作)(其他n-1个事务的动作)

➤两段封锁协议可以保证冲突可串行化。归纳完毕。

仔细阅读一下此证明过程，加强理解

两段封锁协议

(3)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
$L_1(A); \text{READ}(A, t)$	
$t := t + 100$	$L_2(B); \text{READ}(B, s)$
	$s := s * 2$
	$L_2(A);$ 拒绝...等待
$L_1(B);$ 拒绝...等待	
...	...

如何判断“死锁”的发生呢？

如何消除“死锁”呢？

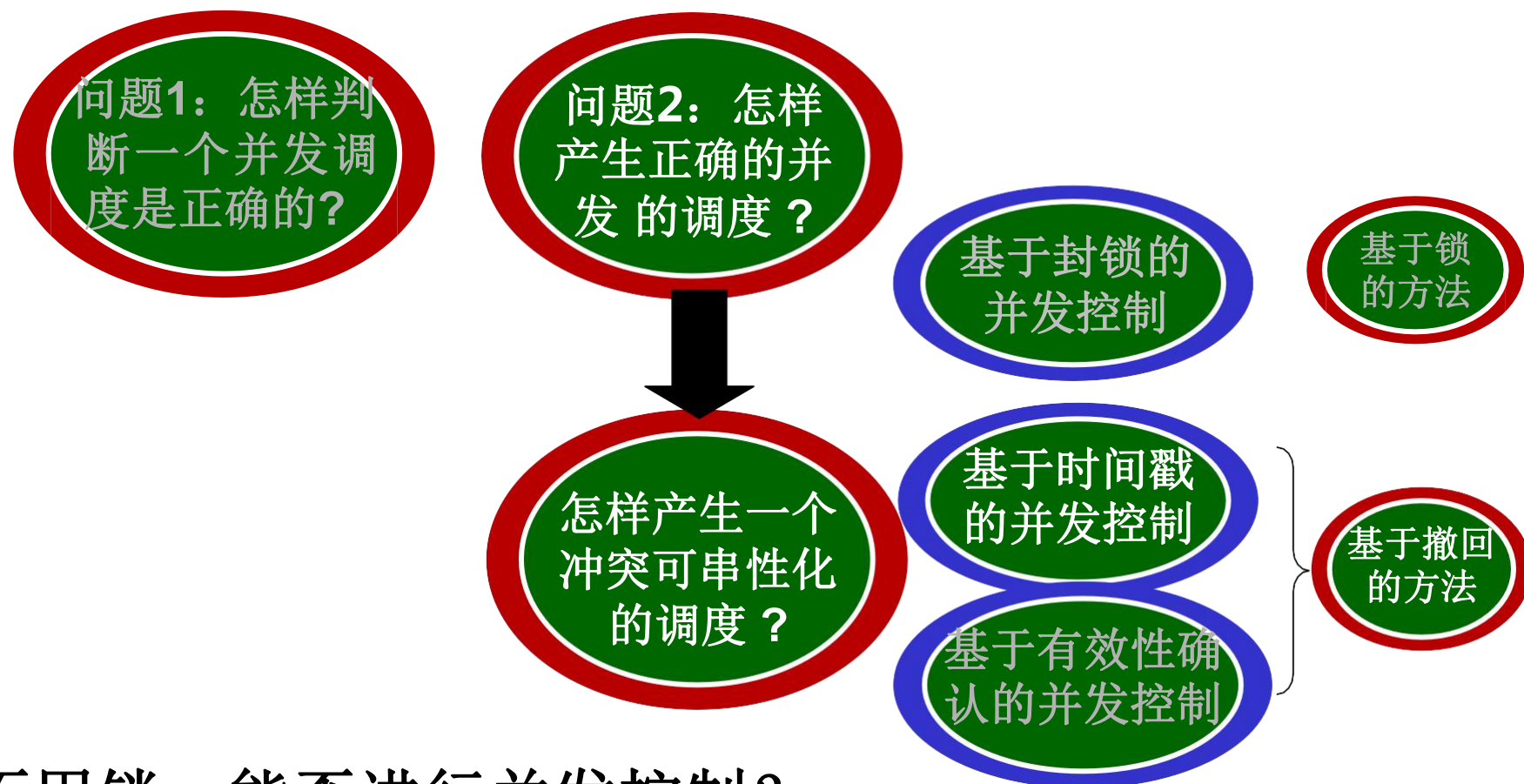
基于时间戳的并发控制方法

基于时间戳的并发控制方法

(1)问题

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。

●并发调度的正确性 \supseteq 可串行性 \supseteq 冲突可串行性



不用锁，能否进行并发控制？

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

基于时间戳的并发控制

➤借助于时间戳，强制使一组并发事务的交叉执行，等价于一个特定顺序的串行执行。

➤特定顺序：时间戳由小到大。

➤如何强制：执行时判断冲突，

□如无冲突，予以执行；

□如有冲突，则撤销事务，并重启该事务，此时该事务获得了一个更大的时间戳，表明是后执行的事务。

➤有哪些冲突：

□读-读无冲突；

□读-写或写-读冲突；

□写-写冲突。

	T1	T2	T3
时间戳	200	150	175
1	r ₁ (B)		
2		r ₂ (A)	
3			r ₃ (C)
4	w ₁ (B)		
5	w ₁ (A)		
6		w ₂ (C)	
7			w ₃ (A)

一种简单的调度规则

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。

事务的时间戳

➤ **TS(T)**: 即**TimeStamp**

一种简单的调度规则

➤读-写并发: (读-写、写-读)

若T事务读x, 则将T的时间戳TS与WT(x)比较:

- ✓若TS大(T后进行), 则允许T操作, 并且更改RT(x)为 $\max\{RT(x), TS\}$;
- ✓否则, 有冲突, 撤回T, 重启T。

若T事务写x, 则将T的时间戳TS与RT(x)比较:

- ✓若TS大(T后进行), 则允许T操作, 并且更改WT(x)为 $\max\{WT(x), TS\}$;
- ✓否则, 有冲突, 撤回T重做。

➤写-写并发

若T事务写x, 则将T的时间戳TS与WT(x)比较:

- ✓若TS大, 则允许T写, 并且更改WT(x)为T的时间戳;
- ✓否则有冲突, T撤回重做。

基于时间戳的并发控制方法

(4)基于时间戳的简单调度规则

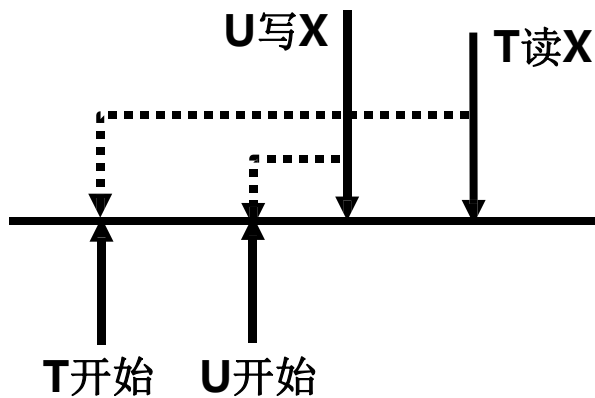
示例

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)				
	撤回T, 重启T2	w ₃ (A)			

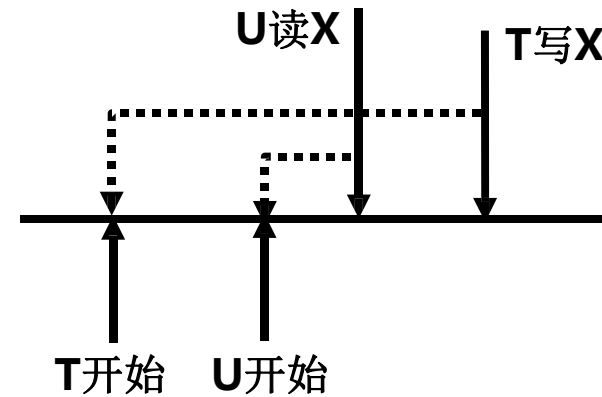
基于时间戳的并发控制方法

(4)基于时间戳的简单调度规则

由前述规则可以解决的...



T过晚的读



T过晚的写

这两种“事实上不可实现的”冲突可以避免

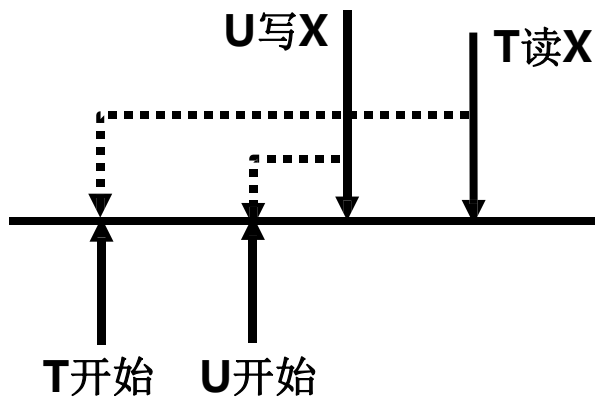
●还有什么问题呢？

基于时间戳的另一种调度规则

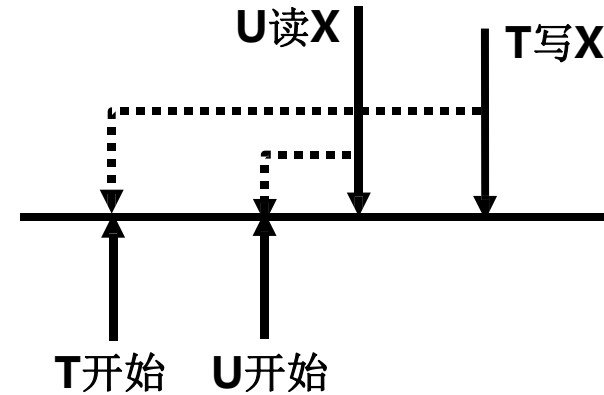
基于时间戳的另一种调度规则

(1)需要解决的问题

由前述简单调度规则可以解决的...



T过晚的读



T过晚的写

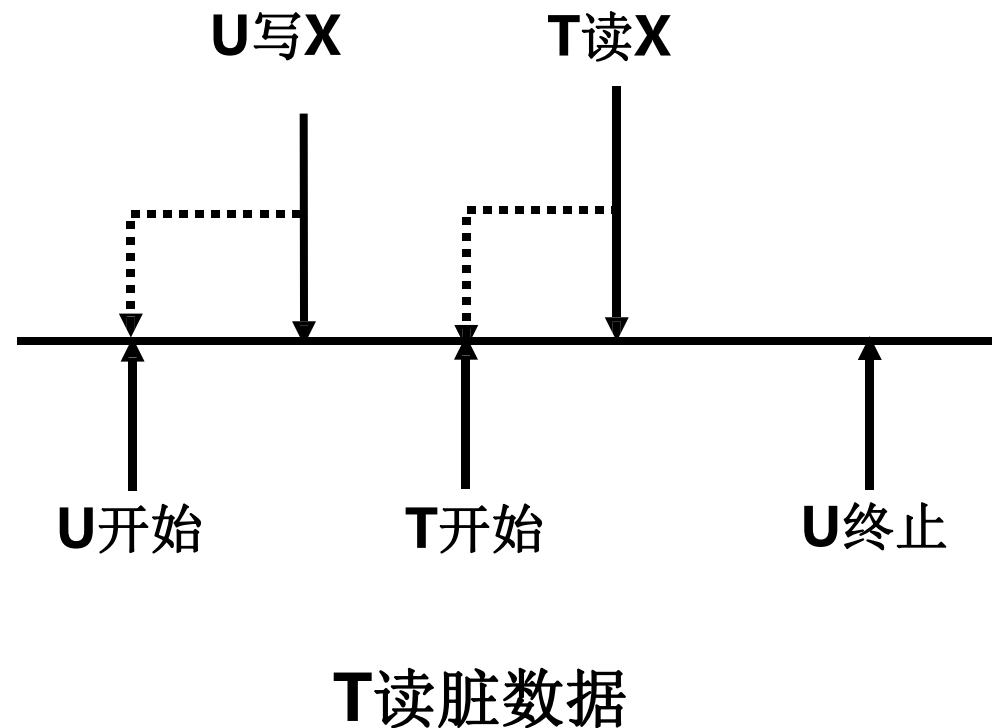
这两种“事实上
不可实现的”冲突
可以避免了

还有什么问题呢？

基于时间戳的另一种调度规则

(1)需要解决的问题

脏读数据如何避免?



基于时间戳的另一种调度规则

(1)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

T1---TS:150

T2---TS:170 **READ(A, t)**

READ(B,s)

s := s*2

t = t*200

WRITE(B,s)

WRITE(B,t)

...

...

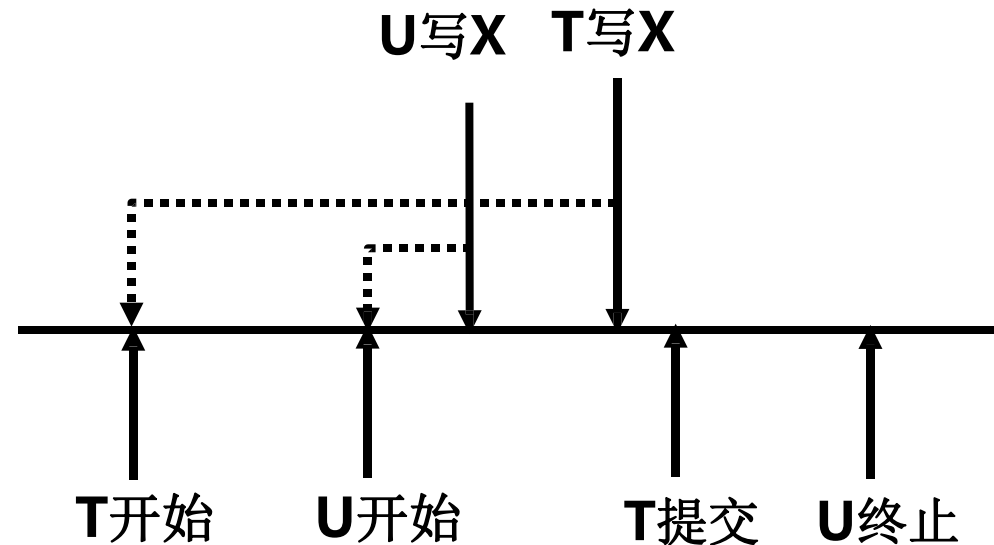
托马斯
写规则

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

(1)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

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。

事务的时间戳

➤ **TS(T)**: 即TimeStamp

如何利用
提交位？

基于时间戳的另一种调度规则

(2)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，被称回滚)
- 推迟T，并在以后决定是终止T还是同意请求(如果请求是读，且此读可能是脏的)

调度规则

➤ 假设调度器收到请求 $r_T(X)$

□(1)如果 $TS(T) \geq WT(x)$, 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

(2)另一种调度规则

➤ 假设调度器收到请求 $w_T(X)$

□(1)如果 $TS(T) \geq RT(x)$, 且 $TS(T) \geq WT(x)$, 此写是事实上是可实现的

✓为 x 写入新值; 置 $WT(x) := TS(T)$; 置 $C(x) := \text{false}$.

□(2)如果 $TS(T) \geq RT(x)$,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

(2)另一种调度规则

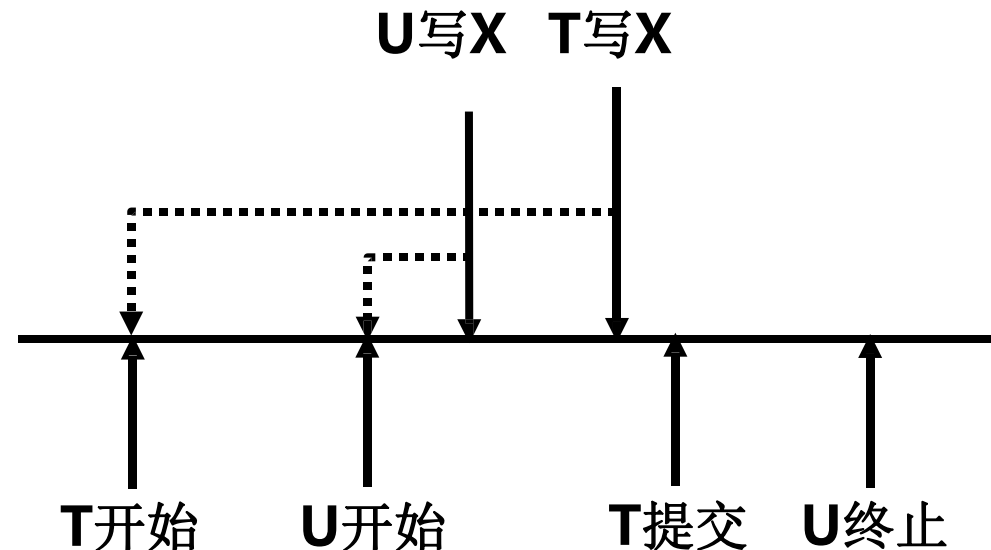
➤假设调度器收到提交T的请求。

□它必须找到T所写的所有数据库元素x, 并置 $C(x) := \text{true}$ 。

□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。



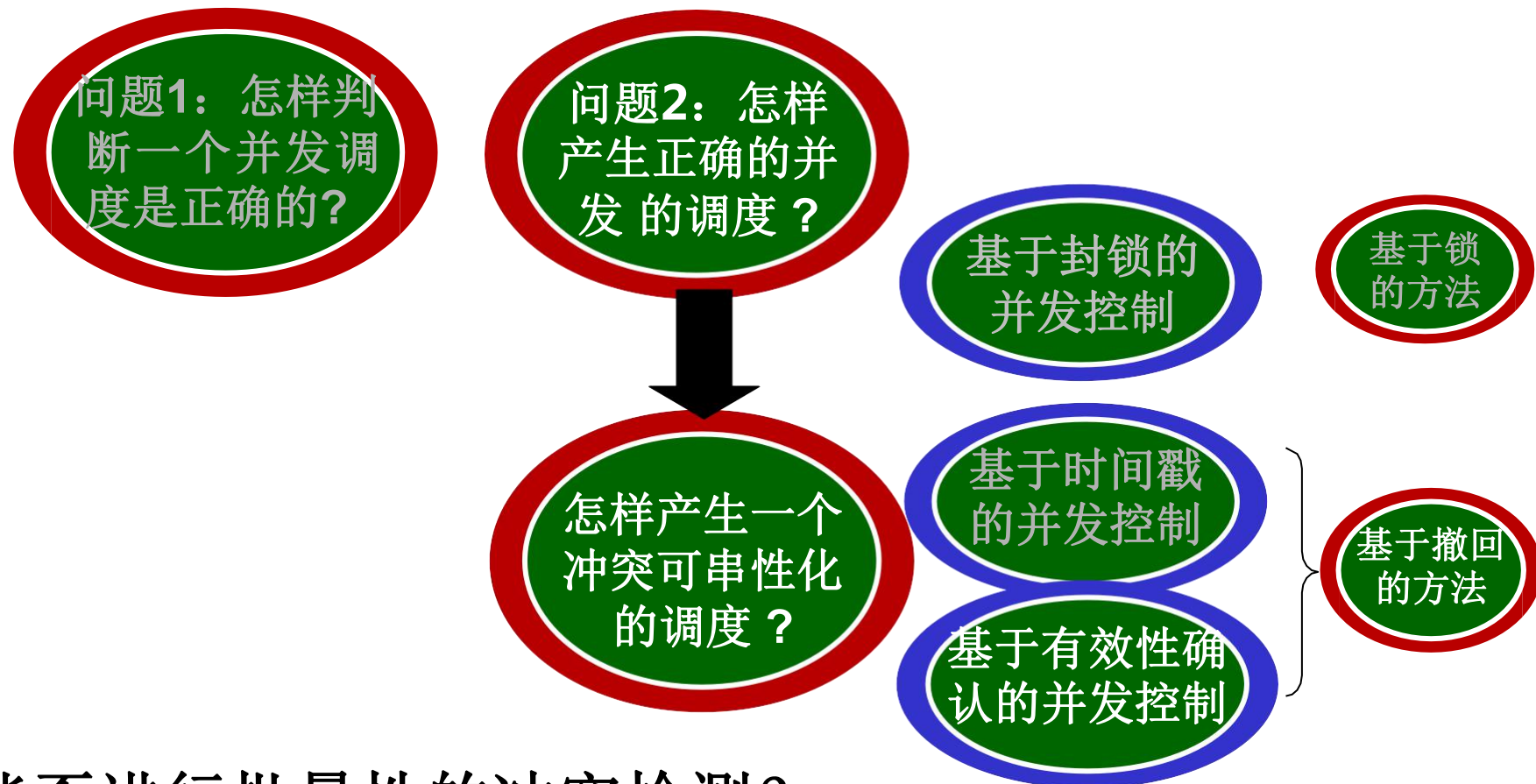
基于有效性确认的并发控制方法

基于有效性确认的并发控制方法

(1)问题？

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。

●并发调度的正确性 \supseteq 可串行性 \supseteq 冲突可串行性



能否进行批量性的冲突检测？

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

(3)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

(4)有效性确认规则？

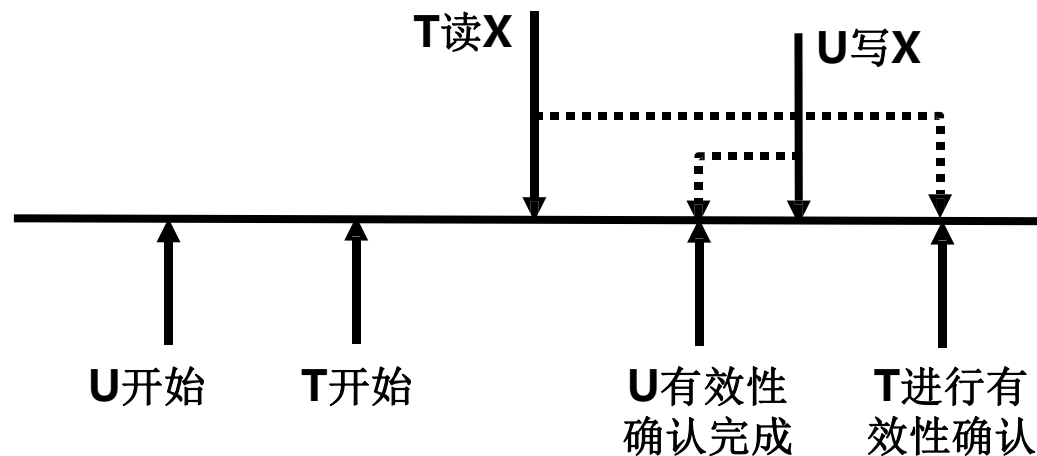
冲突一：假设存在事务U和T满足：

(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) $RS(T) \cap WS(U)$ 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

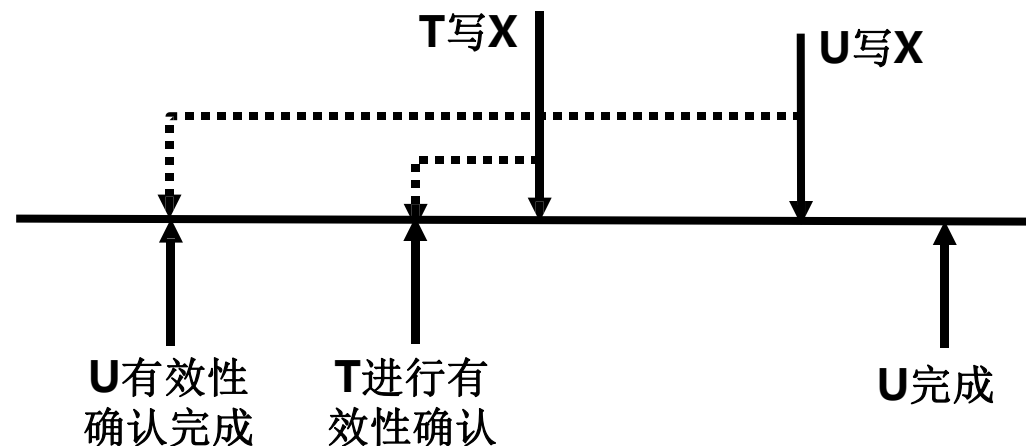
(4)有效性确认规则？

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) $WS(T) \cap WS(U)$ 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

(4)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:

$RS(T) \cap WS(U)$ 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:

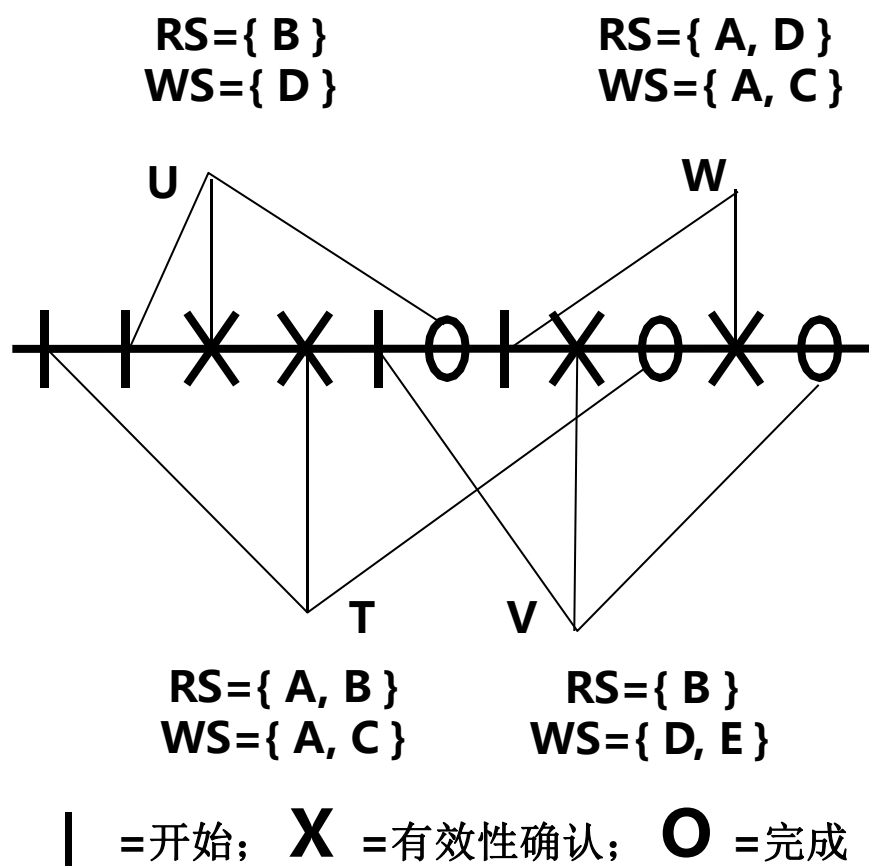
$WS(T) \cap WS(U)$ 是否为空。

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

(4)有效性确认规则？

示例：确认下列四个事务的有效性



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 $\text{FIN}(U) > \text{START}(T)$, 需检测 $\text{RS}(T) \cap \text{WS}(U)$

因 $\text{FIN}(U) > \text{VAL}(T)$, 需检测 $\text{WS}(T) \cap \text{WS}(U)$

检测结果：均为空，则确认T。

3. V的有效性确认

因 $\text{FIN}(U) > \text{START}(V)$, 需检测 $\text{RS}(V) \cap \text{WS}(U)$

因 $\text{FIN}(T) > \text{START}(V)$, 需检测 $\text{RS}(V) \cap \text{WS}(T)$

因 $\text{FIN}(T) > \text{VAL}(V)$, 需检测 $\text{WS}(T) \cap \text{WS}(V)$

检测结果：均为空，则确认V。

4. W的有效性确认

因 $\text{FIN}(T) > \text{START}(W)$, 需检测 $\text{RS}(W) \cap \text{WS}(T)$

因 $\text{FIN}(V) > \text{START}(W)$, 需检测 $\text{RS}(W) \cap \text{WS}(V)$

因 $\text{FIN}(V) > \text{VAL}(W)$, 需检测 $\text{WS}(V) \cap \text{WS}(W)$

检测结果：不全为空, 则W不能确认, W被回滚。

回顾本讲学了什么？

回顾本讲学习了什么？

