第8章之一
数据库事务处理技术
(并发控制)

为什么要进行并发控制

如果不进行并发控制，会怎么样?

如果不进行并发控制，会怎么样?


2BRI564
California
L300
V6
SE

大规模高并发场景

- 双十一
- 电商秒杀抢购
- 过年购买火车票
- 房产在线开盘
- 股票交易
- 明星娱乐事件
- 。。。

会遇到什么问题?

- **宕机**
- **交易出错**
- **金钱损失**
- **客户流失**
- **。。。**

解决方案？

解决方案

- ✓限流
  - ✓校验
  - ✓漏桶算法、令牌算法
- ✓缓存
  - ✓热点数据
- ✓异步
  - ✓立即响应
  - ✓异步处理后续步骤
- ✓分流

解决方案

- ✓数据库系统
  - ✓集群策略→一台宕机不影响整个系统；
  - ✓负载均衡
  - ✓读写分离、分库分表→降低数据库负载
- ✓分布式缓存系统
- ✓监控系统
  - ✓日志记录，溯源
- ✓前端系统
  - ✓智能镜像网站+缓存
- ✓CPU与IO的平衡
- ✓。。。

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?

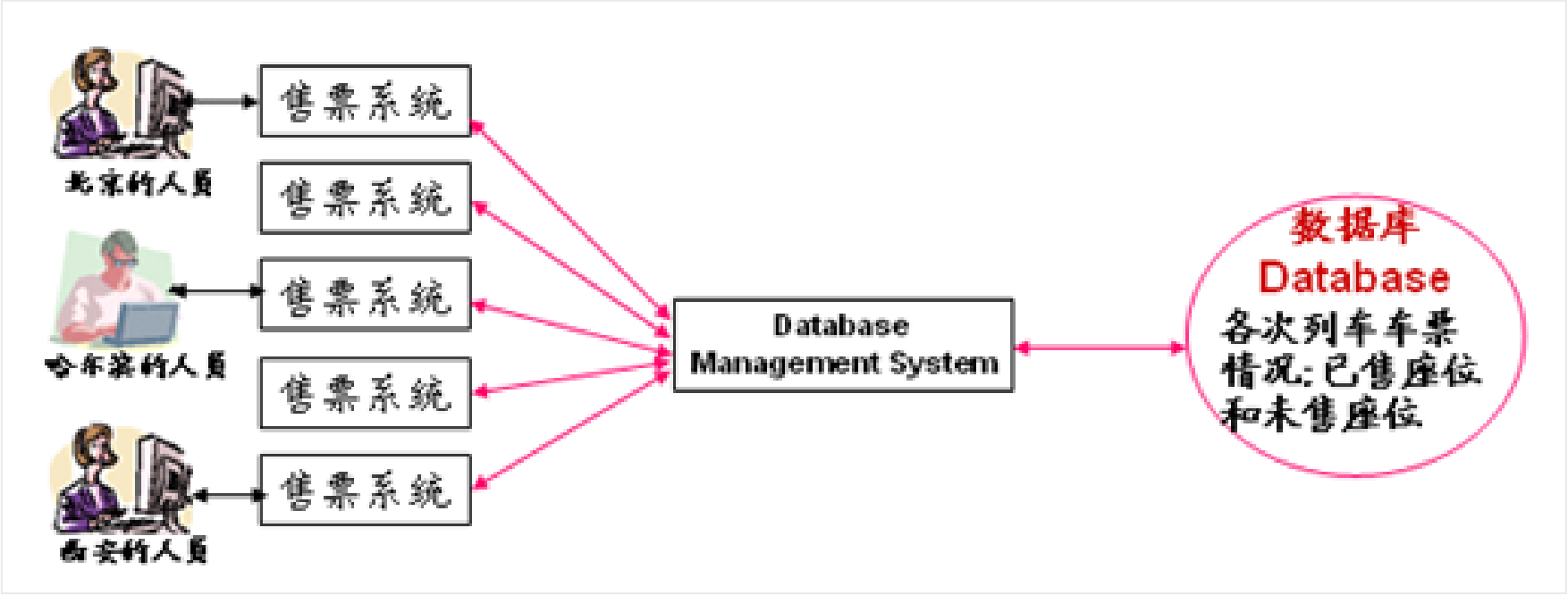
重点与难点

- 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?

(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

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。

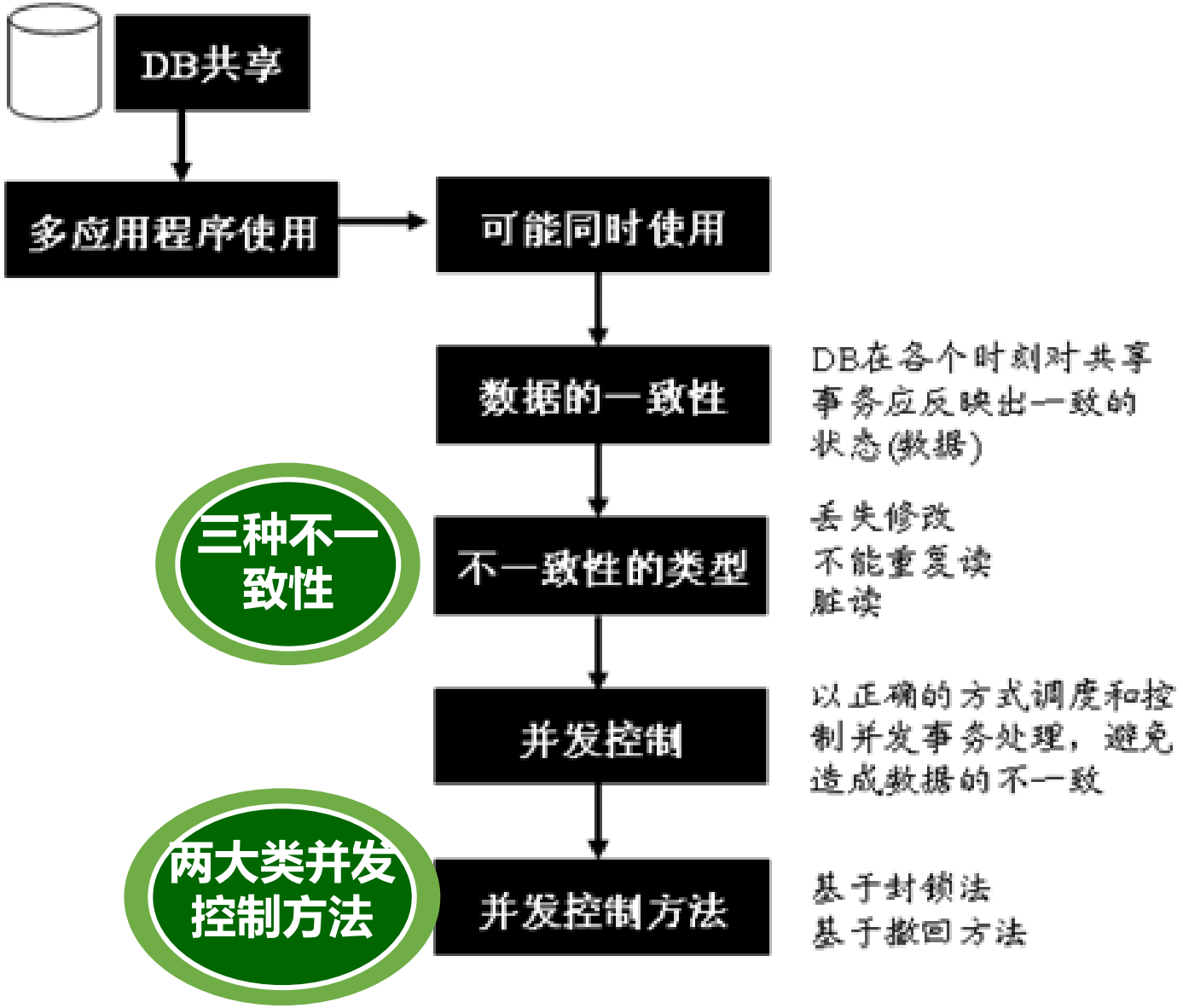
3. 脏读

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。

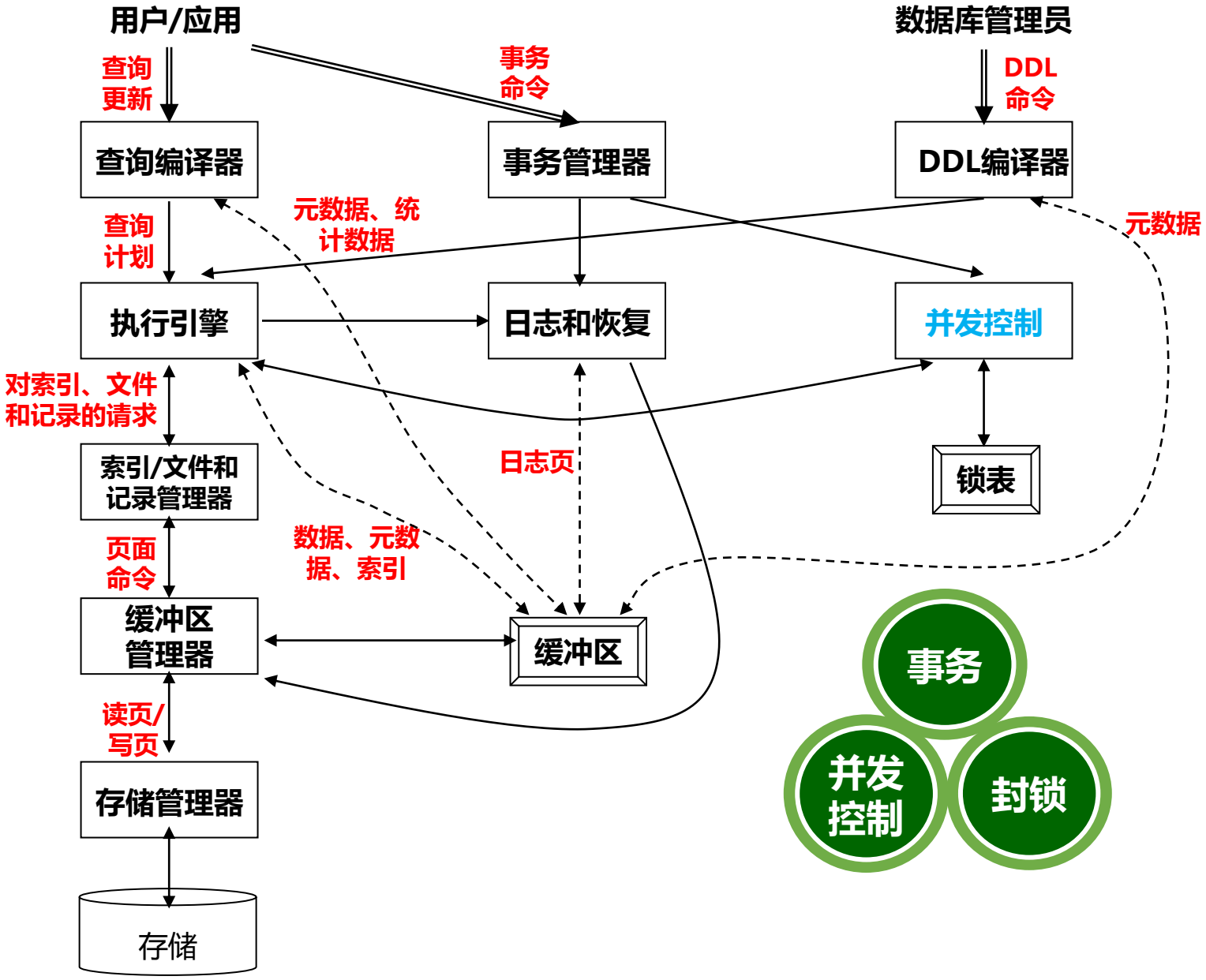
为什么要进行并发控制?

(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

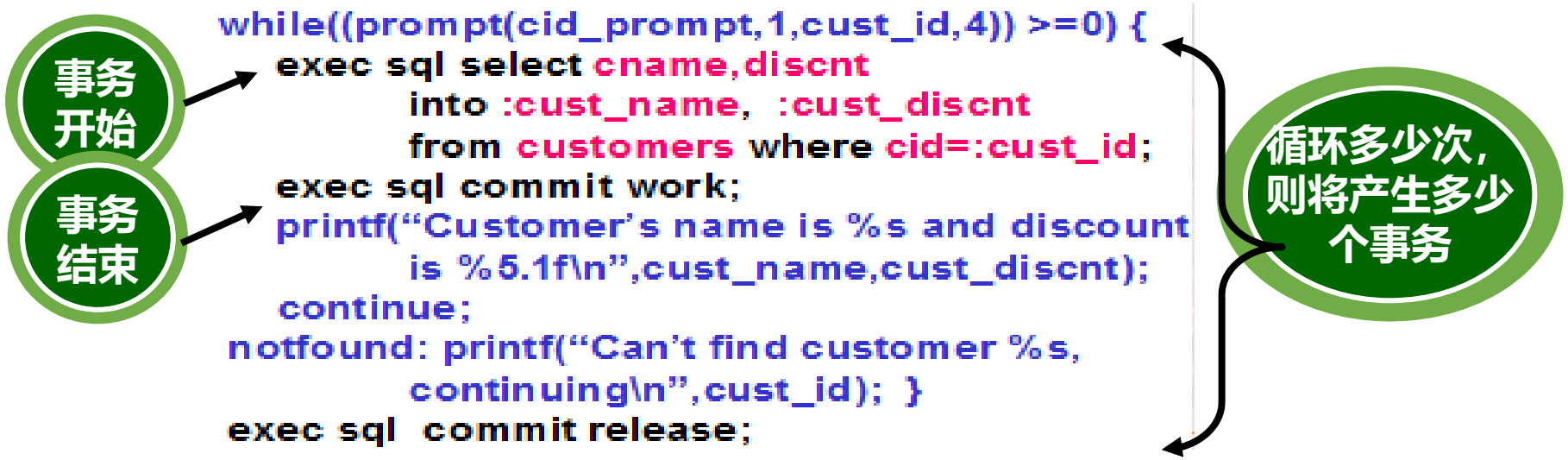
- 一个事务可处理一个数据或一条记录，如下例：

  “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)	

$A_1 \neq A$

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 A_1。

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

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)	

$A_1 \neq A_2$ 两次读的不是同一数据。

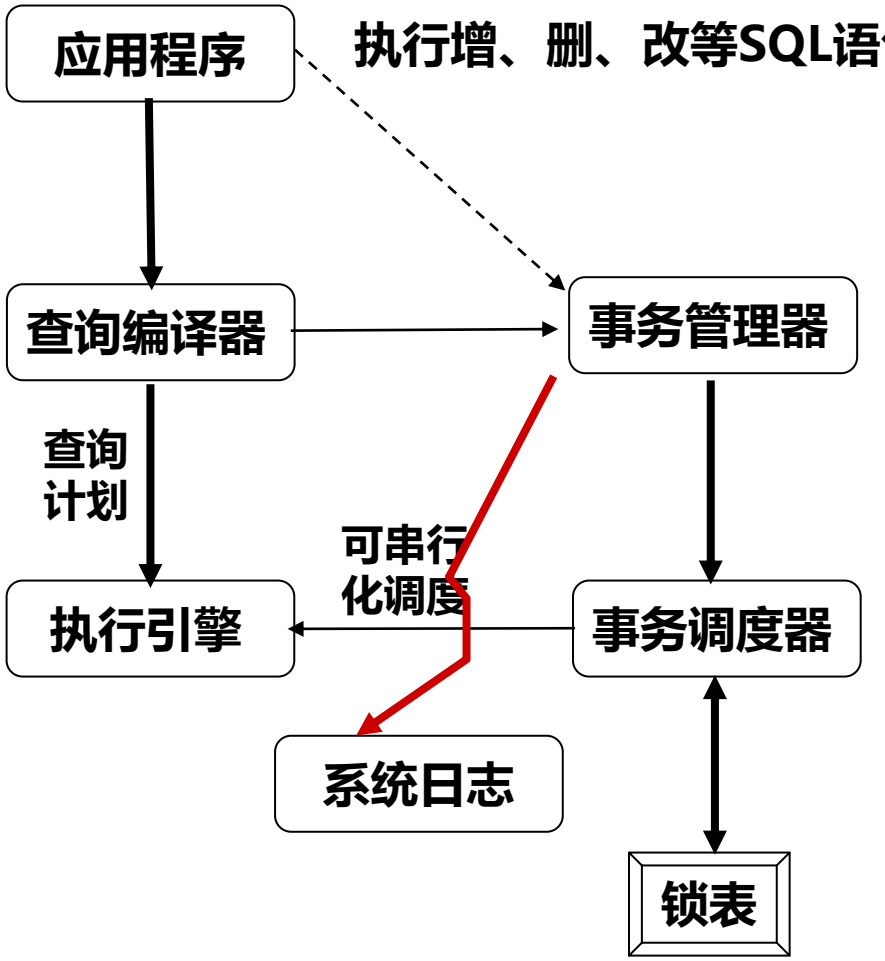
3. 脏读

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。

什么是事务

(4)DBMS对事务的控制



- 产生事务
- 管理事务的时间戳
- 管理事务的一系列操作

- 对所有事务的操作产生一个**读写操作序列(即调度)**
- 保证事务的一致性。

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

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。

上菜次序

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	
		C=C+20
		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。

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

事务调度与可串行性

(3)冲突可串行性

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

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

  - ✓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

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

事务调度与可串行性

(3)冲突可串行性

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。

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

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

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

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

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

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。

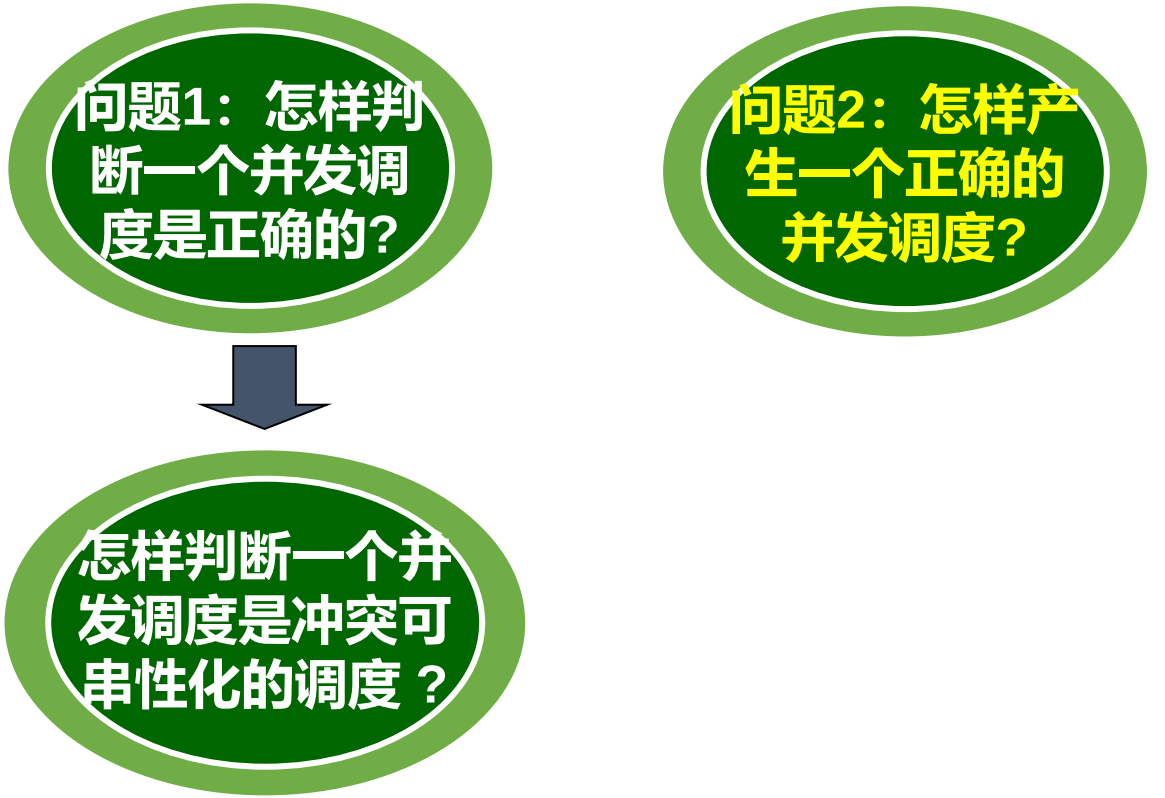
冲突可串行性判别算法

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 ⊇ 可串行性 ⊇ 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, 表示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!**

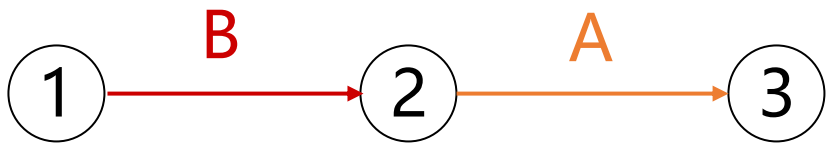
冲突可串行性判别算法(3)示例

示例

- ✓ 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，表示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!

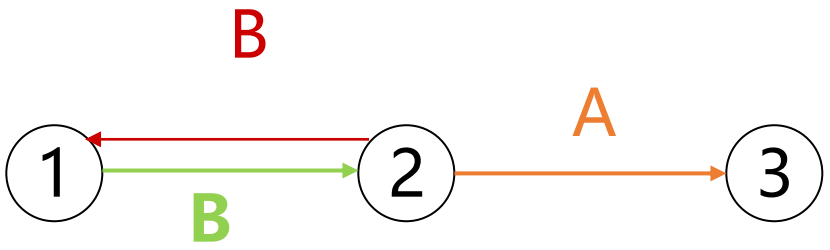
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

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 ⊇ 可串行性 ⊇ 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

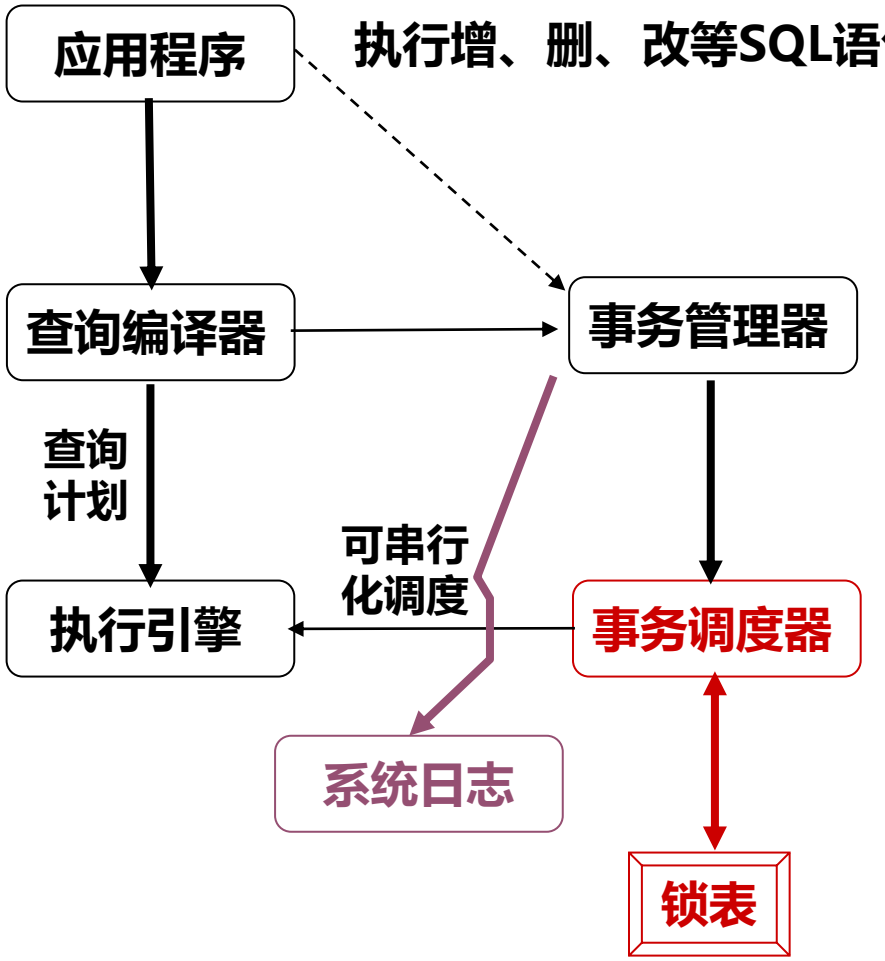
- **每一数据元素都有一唯一的锁**
- **每一事务读写数据元素前，要获得锁。**
- **如果被其他事务持有该元素的锁，则要等待。**
- **事务处理完成后要释放锁。**

$L_i(A)$ ：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?



- 产生事务
- 管理事务的时间戳
- 管理事务的一系列操作

- 对所有事务的操作产生一个读写操作序列(即调度)
- 保证事务的一致性。

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
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);

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

封锁协议之锁的类型

- **排他锁X** (eXclusive 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** (Shared locks)

  所有事务都可以读，但任何事务都不能写

- **更新锁U** (Update locks)

  初始读，以后可升级为写

- **增量锁I** (Incremental 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？

- 排他锁X (eXclusive locks)
- 共享锁S (Shared locks)
- 更新锁U (Update locks)
- 增量锁I (Incremental lock)

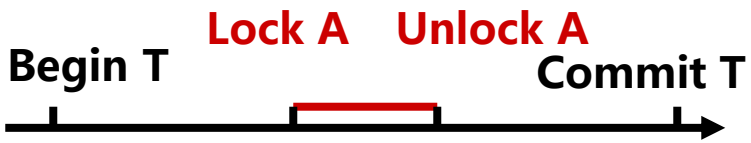
这只是简单形式，还有更丰富内容

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)

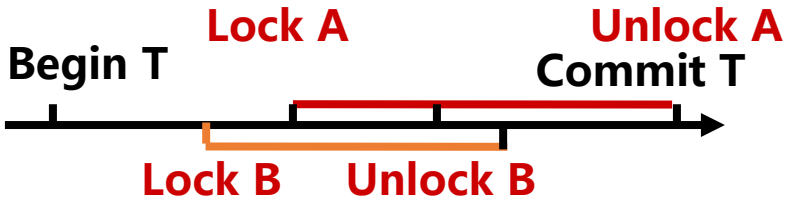


有写要求的数据对象A加排他锁，不再访问后即刻解锁。可防止丢失修改，但允许脏读，允许重复读错误

1级协议(1-LP)

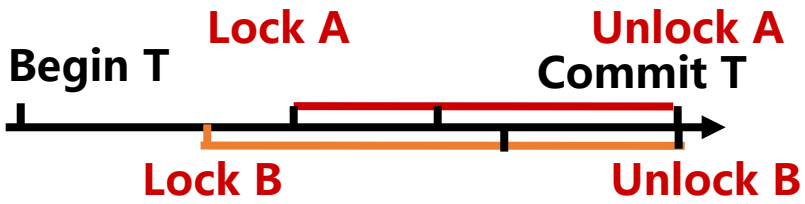
有写要求的数据对象A加排他锁，事务提交时刻解锁。可防止丢失修改，可恢复，防止脏读，允许重复读错误

2级协议(2-LP)



有写要求的数据对象A加排他锁，事务提交时刻解锁。有读要求的数据对象B加共享锁，不再访问后即刻解锁。可防止丢失修改，防止脏读，不允许重复读错误。

3级协议(3-LP)



有写要求的数据对象A加排他锁，事务提交时刻解锁。有读要求的数据对象B加共享锁，事务提交时刻解锁。防止所有不一致性。(如幻读)。

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。

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 ⊇ 可串行性 ⊇ 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 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
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

T_1事务在$U_1(A)$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

(1)什么是两段封锁协议?

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
$L_1(A)$;	$L_2(A)$;
$L_1(B)$;	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);	$U_2(A)$;
$U_1(B)$;	$U_2(B)$;

产生调度并执行：$L_1(A)$在$L_2(A)$前执行

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

如果$L_2(A)$在$L_1(A)$前执行是怎样的呢?

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); READ(A, t)	
	L_2(B); READ(B,s)
t := t+100	
	s := s*2
	L_2(A); **拒绝...等待**
L_1(B); **拒绝...等待**	
...	...

*参见《操作系统》课程

死锁的检测处理

1. 死锁的检测与解除

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
- 超时法实现简单，但其不足也很明显
- 一是有可能**误判死锁**，事务因为其他原因使等待时间超过时限，系统会误认为发生了死锁
- 二是时限若设置得**太长**，死锁发生后不能及时发现

死锁的检测处理

2) 等待图法

- 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T1等待T2,则T1、T2之间划一条有向边，从T1指向T2
- 事务等待图动态地反映了所有事务的等待情况。
- 并发控制子系统周期性地（比如每隔1分钟）检测事务等待图，如果发现图中存在回路，则表示系统中出现了死锁。

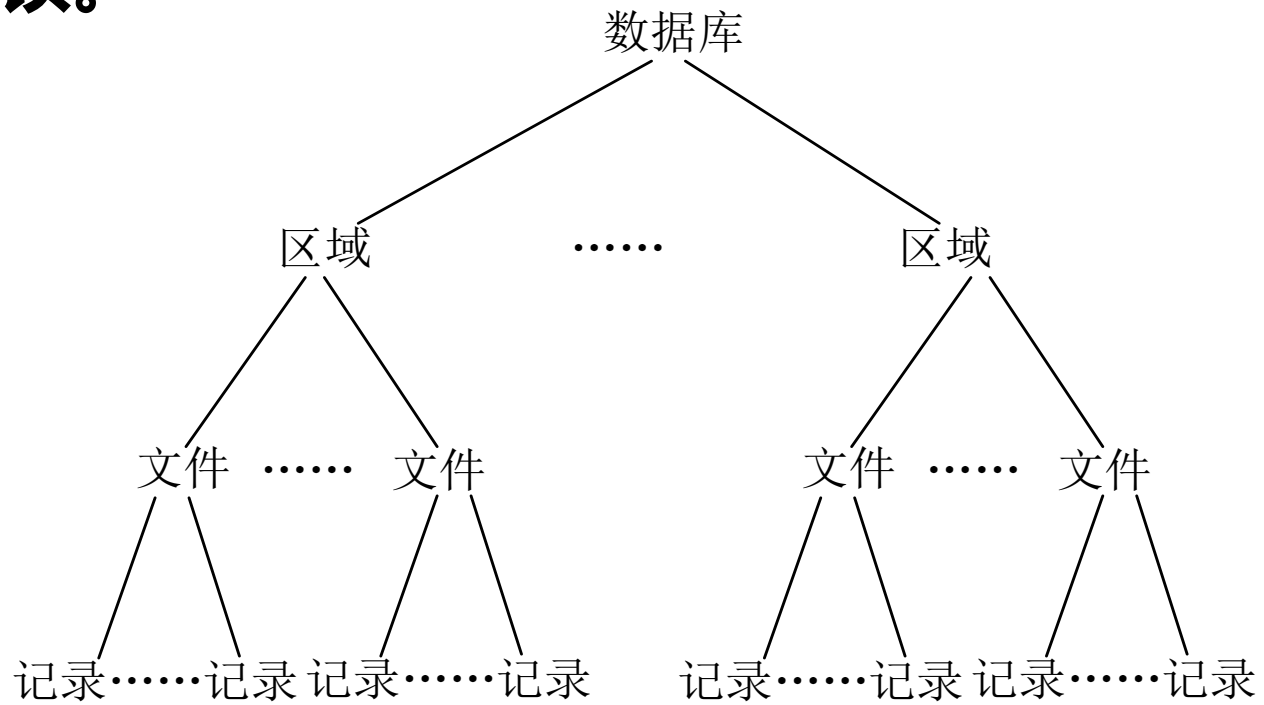
死锁的检测处理

3) 死锁的解除

- 死锁发生后，靠事务自身无法解除，必须由DBMS干预
- 通常采用的方法是选择一个**处理死锁代价最小的事务**，将其撤消，释放此事务持有的所有的锁，使其它事务得以继续运行下去
- 当然，对撤消的事务所执行的数据修改操作必须加以恢复

多粒度封锁

- 多粒度封锁协议允许多粒度树中的每个结点被独立地加锁。
- 对一个结点加锁意味着这个结点的所有后裔结点也被加以同样类型的锁。



多粒度封锁

- 申请封锁时应该按自上而下的次序进行
- 释放封锁时则应该按自下而上的次序进行
- 多粒度封锁方法提高了系统的并发度，减少了加锁和解锁的开销，它已经在实际的数据库管理系统产品中得到广泛应用
- 例如，IBM DB2 UDB DBMS中就采用了这种封锁方法

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 ⊇ 可串行性 ⊇ 冲突可串行性

不用锁，能否进行并发控制?

基于时间戳的并发控制方法

(2)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

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			$w_3(A)$

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

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

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

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
时间戳 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T2，重启T2	$w_3(A)$			

读时间戳

写时间戳

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 ⊇ 可串行性 ⊇ 冲突可串行性

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

(2)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?

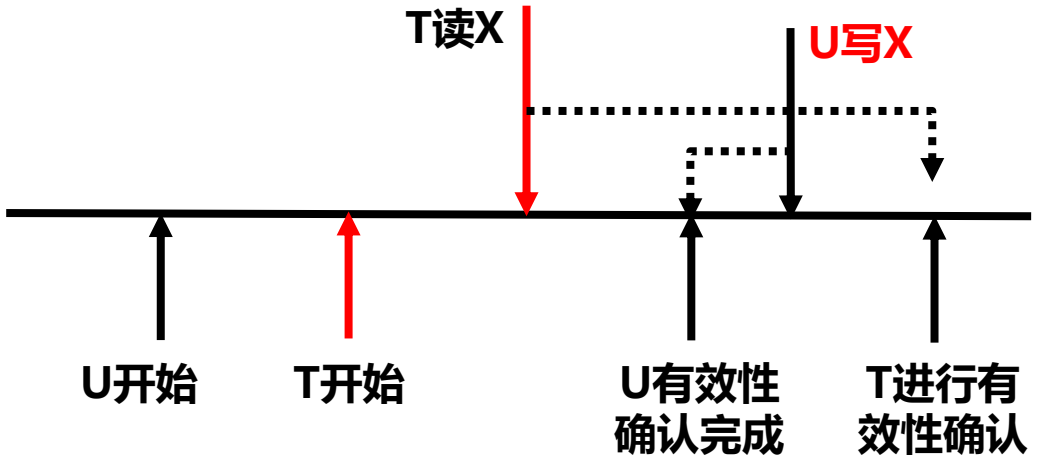
冲突一：假设存在事务U 和 T满足：

(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)∩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?

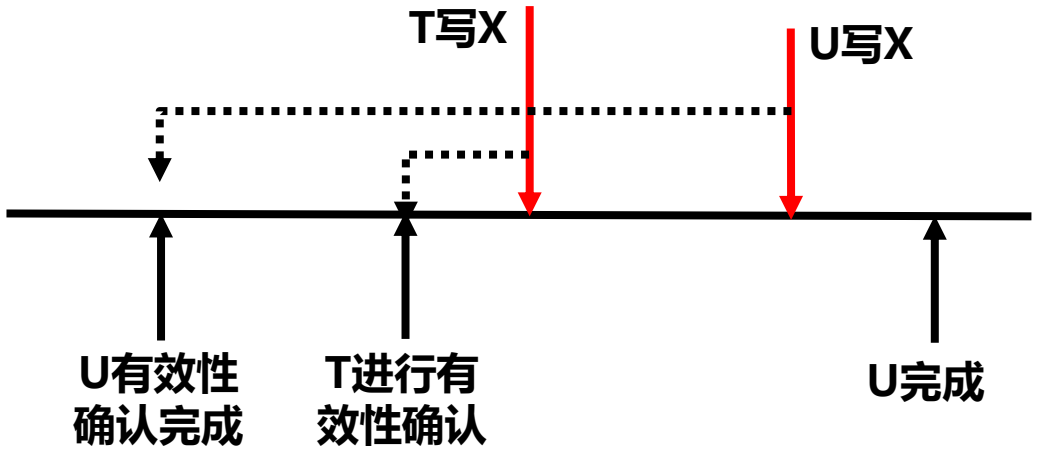
冲突二：假设存在事务U 和T 满足：

(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)∩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) ∩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:

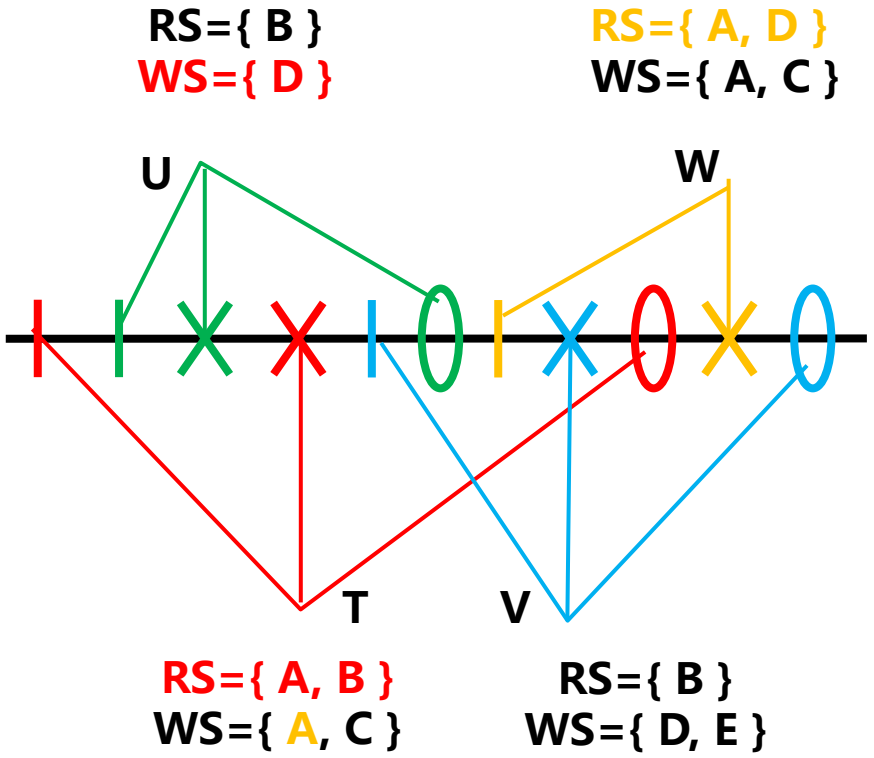
WS(T) ∩WS(U)是否为空。

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

(4)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)

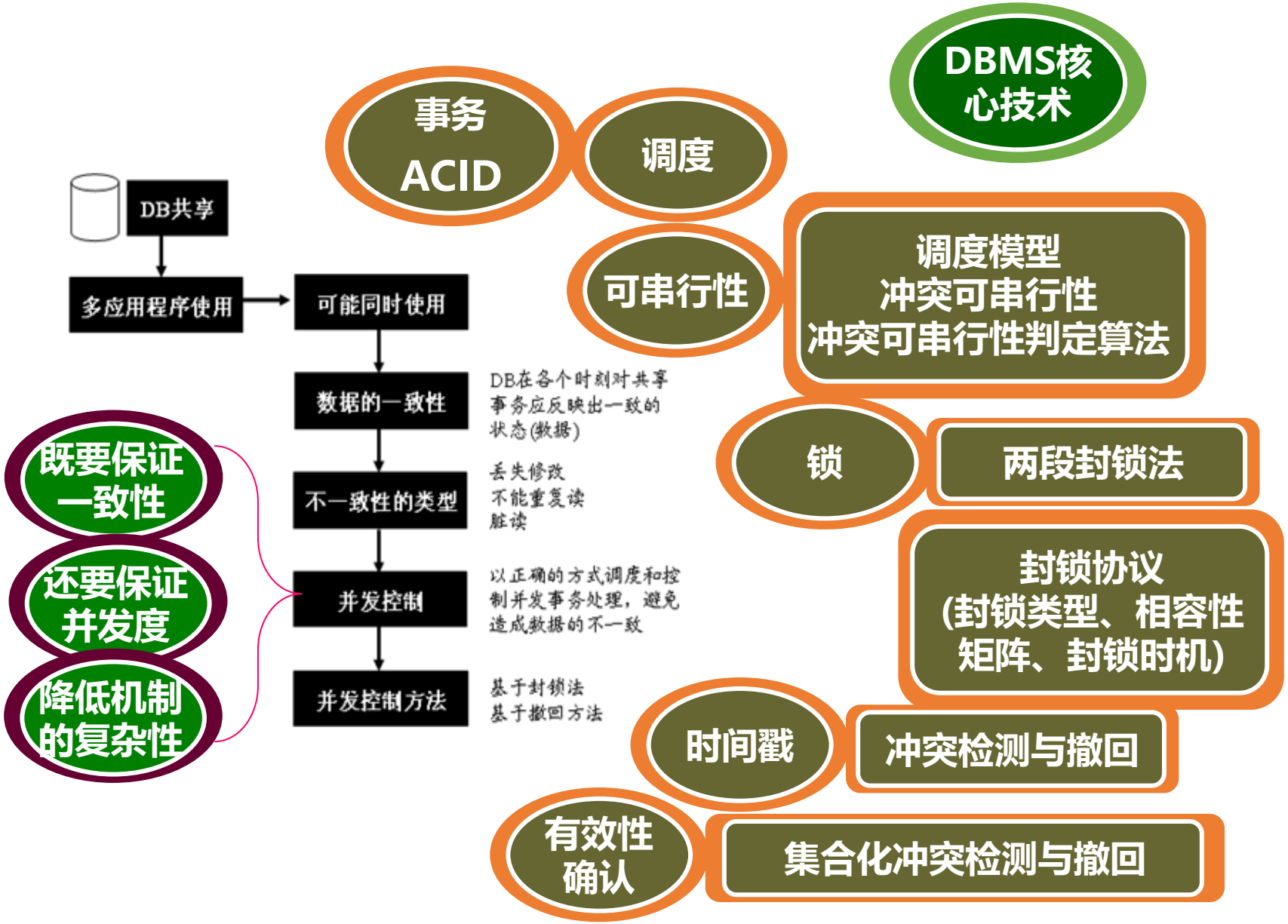
因FIN(V)>START(W),需检测RS(W)∩WS(V)

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。

回顾本讲学了什么?

回顾本讲学习了什么?


DBMS核心技术
事务
ACID
调度
DB共享
多应用程序使用
可能同时使用
数据的一致性
DB在各个时刻对共享事务应反映出一致的状态(数据)
不一致性的类型
丢失修改
不能重复读
脏读
并发控制
以正确的方式调度和控制并发事务处理，避免造成数据的不一致
并发控制方法
基于封锁法
基于撤回方法
既要保证一致性
还要保证并发度
降低机制的复杂性
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

1、设有下面4个事务:

T1: R11(x)W12(x)R13(y)W14(y)R15(z)W16(z)

T2: R21(y)R22 (x)R23(z)W24(x)R25(y)W26(z)

T3: R31(x)R32(y)R33(z)W34(x)

T4: R41(x)R42(y)R43(z)W44(y)

S1, S2 为对 T_1，T_2，T_3，T_4的两个调度:

S1: $R_{11}(x)W_{12}(x)R_{31}(x)R_{13}(y)W_{14}(y)R_{32}(y)R_{21}(y)R_{15}(z)W_{16}(z)$ $R_{33}(z)W_{34}(x)$

$R_{22}(x)R_{23}(z)W_{24}(x)R_{41}(x)$ $R_{42}(y)R_{25}(y)$ $W_{26}(z)R_{43}(z)W_{44}(y)$

S2: $R_{21}(y)R_{11}(x)W_{12}(x)R_{22}$ $(x)R_{23}(z)R_{41}(x)W_{24}(x)$ $R_{13}(y)W_{14}(y)R_{25}(y)W_{26}(z)$

$R_{31}(x)$ $R_{42}(y)$ $R_{15}(z)$ $R_{32}(y)R_{33}(z)W_{16}(z)R_{43}(z)W_{44}(y)$ $W_{34}(x)$

(1) 试画出调度S1和S2的前趋图;

(2) 试判别S1和S2是否为冲突可串行的调度;

2、如图(a)、(b)所示事务T1，T2的并发执行。

T1	T2
read(A)	
A=A+10	
	read(A)
	A=A*3
write(A)	
	write(A)

(a)

T1	T2
read(A)	
A=A+10	
	read(A)
	A=A*3
	write(A)
write(A)	

(b)

设A的初值为10，请分别回答以下问题：

（1）采用具有wait-die策略的两段封锁执行图(a)中事务，说明执行过程及A值最终结果；

（2）采用具有wound-wait策略的两段封锁执行图(b)中事务，说明执行过程及A值最终结果。

3、设T1,T2,T3是如下的三个事务：

T1：A:=A+2;

T2：A:=A×2;

T3：$A:=A^2$;

设A的初值为0。

（1）若这三个事务允许并发执行，则有多少种可能的正确结果，请一一列举出来。

（2）请给出一个可串行化的调度，并给出执行结果。

（3）请给出一个不可串行化的调度，并给出执行结果。