第十一章 并发控制

章成源

湖南大学-信息科学与工程学院-计算机科学系

办公室：院楼403

Email：cyzhangcse@hnu.edu.cn

并发控制

- 多用户数据库系统

  允许多个用户同时使用的数据库系统

  - 飞机定票数据库系统
  - 银行数据库系统
  - 特点：在同一时刻并发运行的事务数可达数百上千个

并发控制

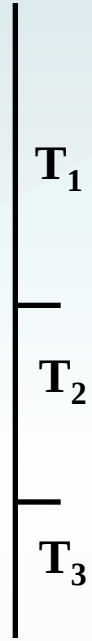
- 数据库管理系统允许多个事务并发执行：
  - 优点
    - 增加系统吞吐量(throughput)。吞吐量是指单位时间系统完成事务的数量。当一事务需等待磁盘I/O时，CPU可去处理其它正在等待CPU的事务。这样，可减少CPU和磁盘空闲时间，增加给定时间内完成事务的数量。
    - 减少平均响应时间(average response time)。事务响应时间是指事务从提交给系统到最后完成所需要的时间。事务的执行时间有长有短，如果按事务到达的顺序依次执行，则短事务就可能会由于等待长事务导致完成时间的延长。如果允许并发执行，短事务可以较早地完成。因此，并发执行可减少事务的平均响应时间。
  - 缺点
    - 若不对事务的并发执行加以控制，则可能破坏数据库的一致性。

并发控制（续）

- 多事务执行方式

（1）事务串行执行

- 每个时刻只有一个事务运行，其他事务必须等到这个事务结束以后方能运行
- 不能充分利用系统资源，发挥数据库共享资源的特点

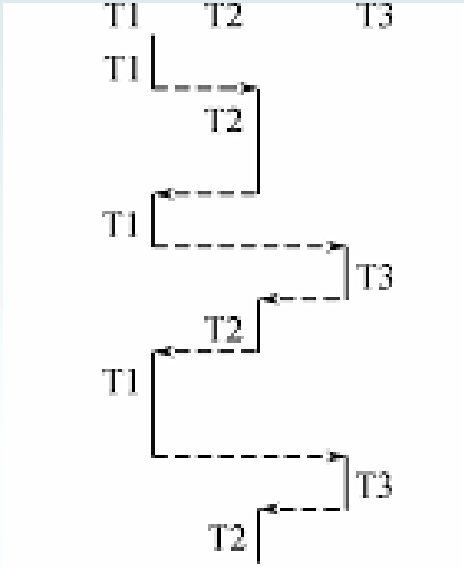


事务的串行执行方式

并发控制（续）

（2）交叉并发方式（Interleaved Concurrency）

- 在单处理机系统中，事务的并行执行是这些并行事务的并行操作轮流交叉运行
- 单处理机系统中的并行事务并没有真正地并行运行，但能够减少处理机的空闲时间，提高系统的效率



(b) 事务的交叉并发执行方式

并发控制（续）

（3）同时并发方式（simultaneous concurrency）

- 多处理机系统中，每个处理机可以运行一个事务，多个处理机可以同时运行多个事务，实现多个事务真正的并行运行
- 最理想的并发方式，但受制于硬件环境
- 更复杂的并发方式机制

并发控制（续）

- 事务并发执行带来的问题
  - 会产生多个事务同时存取同一数据的情况
  - 可能会存取和存储不正确的数据，破坏事务隔离性和数据库的一致性
- 数据库管理系统必须提供并发控制机制
- 并发控制机制是衡量一个数据库管理系统性能的重要标志之一

第11章 并发控制

- 11.1 两阶段封锁协议
- 11.2 时间戳排序协议
- 11.3 乐观并发控制协议
- 11.4 三级封锁协议
- 11.5 多版本并发控制

并发控制技术的核心目标

- 万变不离其宗
  - 并发控制的目的，是保证事务的一致性（C）和隔离性（I）
  - 保障一致性和隔离性的核心是**可串行化调度**。
- 核心的目标：可串行化调度
  - 多个事务的一个并发执行序列是正确的，当且仅当其结果与按**某一串行**执行这些事务时的结果相同

并发控制技术的核心目标

- 并发控制技术的主要思想：**先定序、后检验**
  - **步骤1：定序。**为这些事务规定一个执行顺序（全序）

  - 不同的并发控制技术区别：顺序如何确定？
  - **步骤2：检验。**每个事务的执行是否按照事先规定的顺序执行的
  - 不同的并发控制技术区别
    - 检验什么、什么时候检验、检验不通过怎么办？

内容提要

- 可串行化下的并发控制
- 弱隔离级别下的并发控制

第11章 并发控制

- 11.1 两阶段封锁协议
- 11.2 时间戳排序协议
- 11.3 乐观并发控制协议
- 11.4 三级封锁协议
- 11.5 多版本并发控制

11.1 两阶段封锁协议

- 11.1.1 两阶段封锁协议概念
- 11.1.2 严格与强严格两阶段封锁协议
- 11.1.3 死锁预防实现技术

两阶段锁协议

- 数据库管理系统普遍采用**两段锁协议的方法**实现并发调度的可串行性，从而保证调度的正确性
- 两阶段锁协议是利用**封锁**来进行并发控制协议的技术
  - 在详细介绍两阶段锁之前，我们首先了解封锁的含义

什么是封锁

- 封锁就是事务T在对某个数据对象（例如表、记录等）操作之前，先向系统发出请求，对其加锁
- 加锁后事务T就对该数据对象有了一定的控制，在事务T释放它的锁之前，其它的事务不能更新此数据对象。
- 封锁是实现并发控制的一个非常重要的技术

基本封锁类型

- 一个事务对某个数据对象加锁后究竟拥有什么样的控制由封锁的类型决定。
- 基本封锁类型
  - 排它锁（Exclusive Locks，简记为X锁）
  - 共享锁（Share Locks，简记为S锁）

排它锁

- 排它锁又称为写锁
- 若事务T对数据对象A加上X锁，则只允许T读取和修改A，其它任何事务都不能再对A加任何类型的锁，直到T释放A上的锁
- 保证其他事务在T释放A上的锁之前不能再读取和修改A

共享锁

- 共享锁又称为读锁
- 若事务T对数据对象A加上S锁，则事务T可以读A但不能修改A，其它事务只能再对A加S锁，而不能加X锁，直到T释放A上的S锁
- 保证其他事务可以读A，但在T释放A上的S锁之前不能对A做任何修改

锁的相容矩阵

T_2 \ T_1	X	S	–
X	N	N	Y
S	N	Y	Y
–	Y	Y	Y

Y=Yes，相容的请求
N=No，不相容的请求

封锁协议

- 什么是封锁协议
  - 在运用X锁和S锁对数据对象加锁时，需要约定一些规则，这些规则为封锁协议（Locking Protocol）。
    - 何时申请X锁或S锁
    - 持锁时间
    - 何时释放
  - 对封锁方式规定不同的规则，就形成了各种不同的封锁协议，它们分别在不同的程度上为并发操作的正确调度提供一定的保证。

两阶段锁协议

- 数据库管理系统普遍采用两段锁协议的方法实现并发调度的可串行性，从而保证调度的正确性
- 两段锁协议

  指所有事务必须分两个阶段对数据项加锁和解锁

  - 在对任何数据进行读、写操作之前，事务首先要获得对该数据的封锁
  - 在释放一个封锁之后，事务不再申请和获得任何其他封锁

两阶段锁协议

- “两段”锁的含义

  事务分为两个阶段

  - 第一阶段是获得封锁，也称为扩展阶段
    - 事务可以申请获得任何数据项上的任何类型的锁，但是不能释放任何锁
  - 第二阶段是释放封锁，也称为收缩阶段
    - 事务可以释放任何数据项上的任何类型的锁，但是不能再申请任何锁

两阶段锁协议

【例1】：事务Ti遵守两段锁协议，其封锁序列是 ：

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

|←　　扩展阶段　　→|　　|←　　收缩阶段　　→|

事务Tj不遵守两段锁协议，其封锁序列是：

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

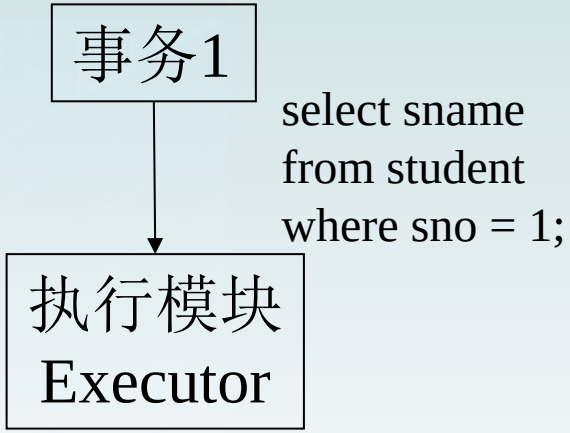
两阶段锁协议

例：

步骤	T1	T2
①	Begin;	Begin;
②	Slock A	
③	R_1(A): A = 100;	
④	Xlock A	
⑤	W_1(A-50)	
⑥		Slock A
⑦	C_1;	等待
⑧	Unlock A	
⑨		R_2(A): A = 50;
⑩		Xlock A
⑪		W_2(A-50)
⑫		C_2;
⑬		Unlock A

- T1的封锁序列：
  - 扩展阶段：Slock A, Xlock A
  - 收缩阶段：Unlock A
- T2的封锁序列：
  - 扩展阶段：Slock A, Xlock A
  - 收缩阶段：Unlock A

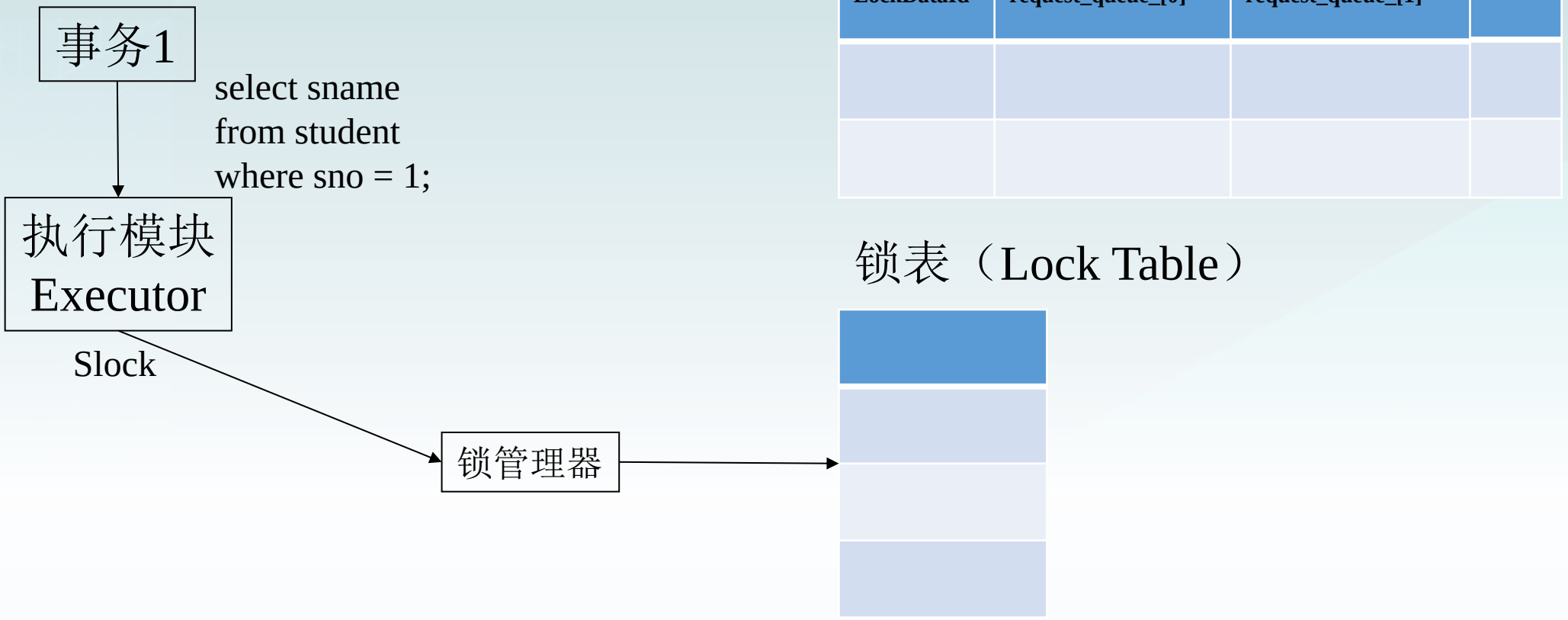
2PL



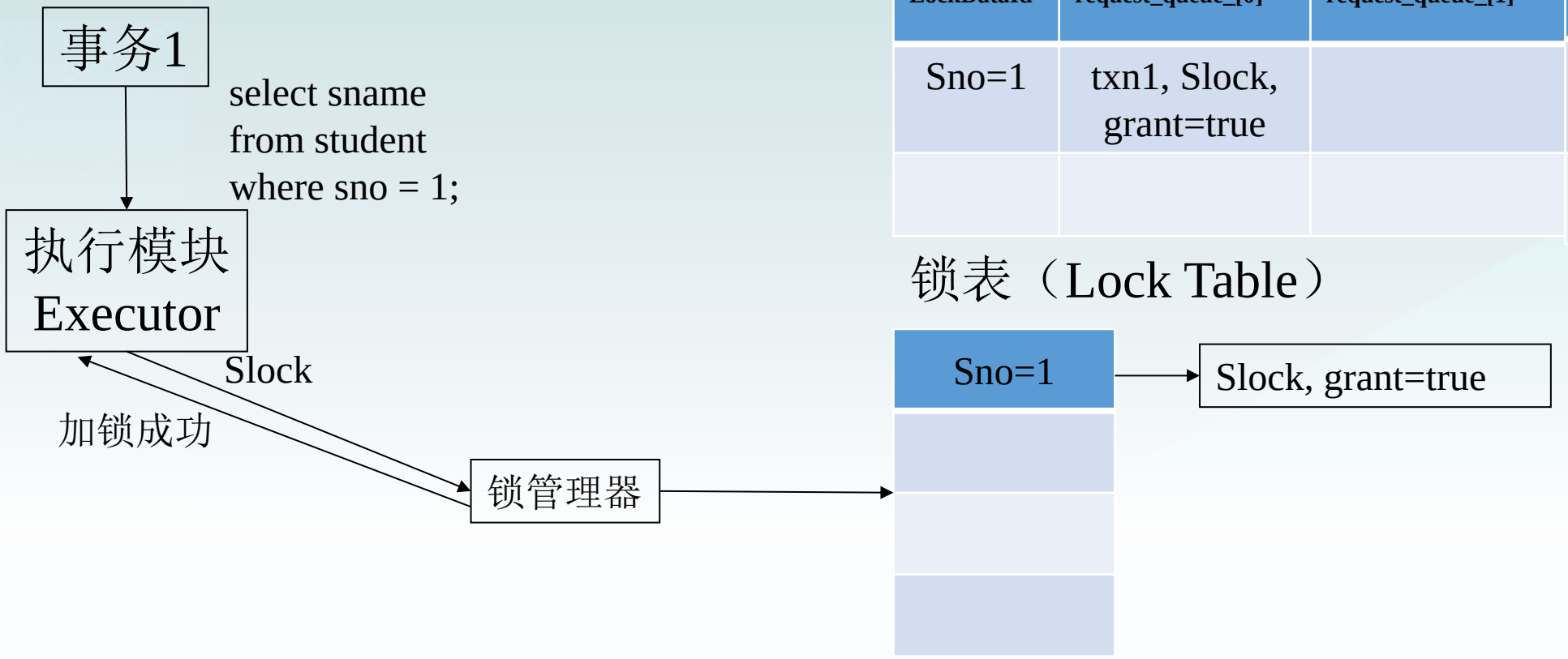
LockDataId	request_queue_[0]	request_queue_[1]	…

锁表（Lock Table）

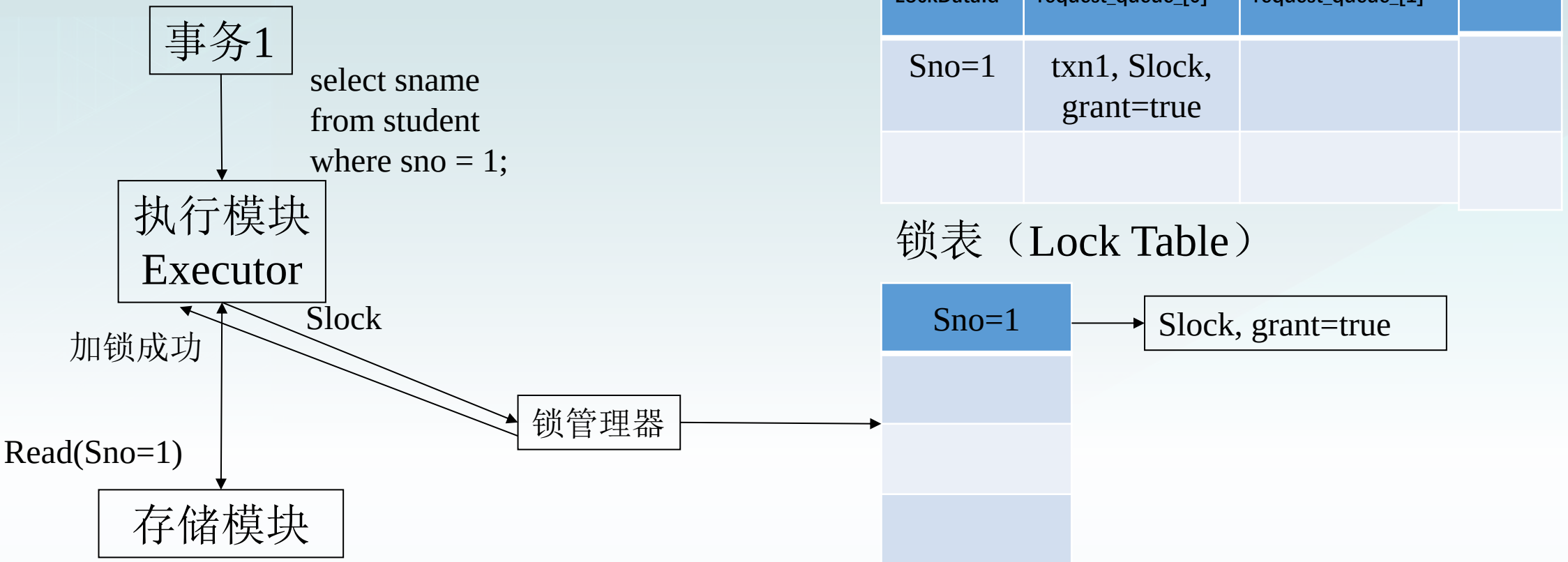
2PL


事务1
select sname
from student
where sno = 1;
执行模块
Executor
Slock
锁管理器
LockDataId
request_queue_[0]
request_queue_[1]
…
锁表（Lock Table）

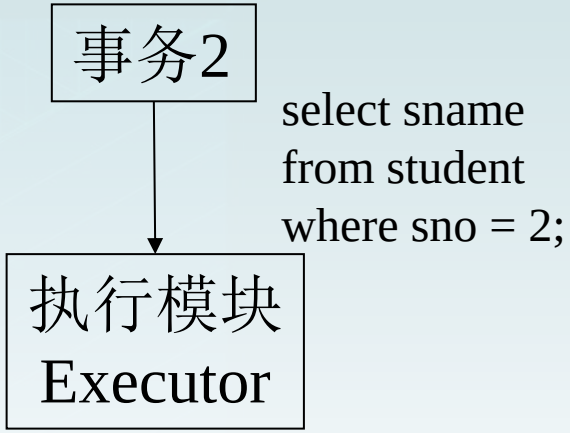
2PL


事务1
select sname
from student
where sno = 1;
执行模块
Executor
Slock
加锁成功
锁管理器
LockDataId
request_queue_[0]
request_queue_[1]
…
Sno=1
txn1, Slock, grant=true
锁表（Lock Table）
Sno=1
Slock, grant=true

2PL

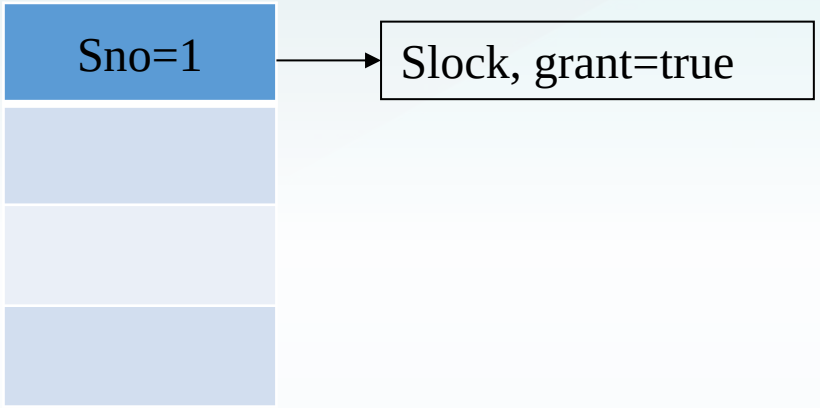

事务1
select sname
from student
where sno = 1;
执行模块
Executor
Slock
加锁成功
Read(Sno=1)
存储模块
锁管理器
LockDataId
request_queue_[0]
request_queue_[1]
…
Sno=1
txn1, Slock, grant=true
锁表（Lock Table）
Sno=1
Slock, grant=true

2PL

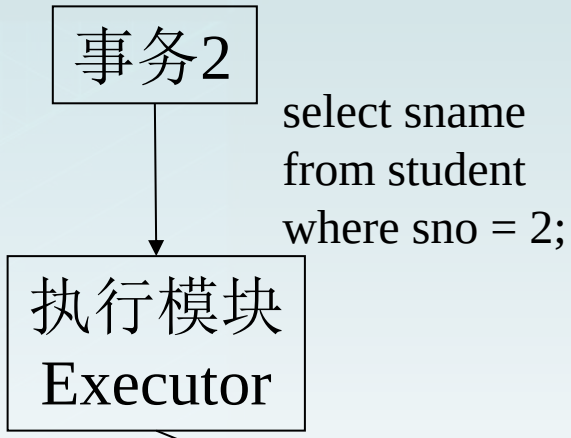


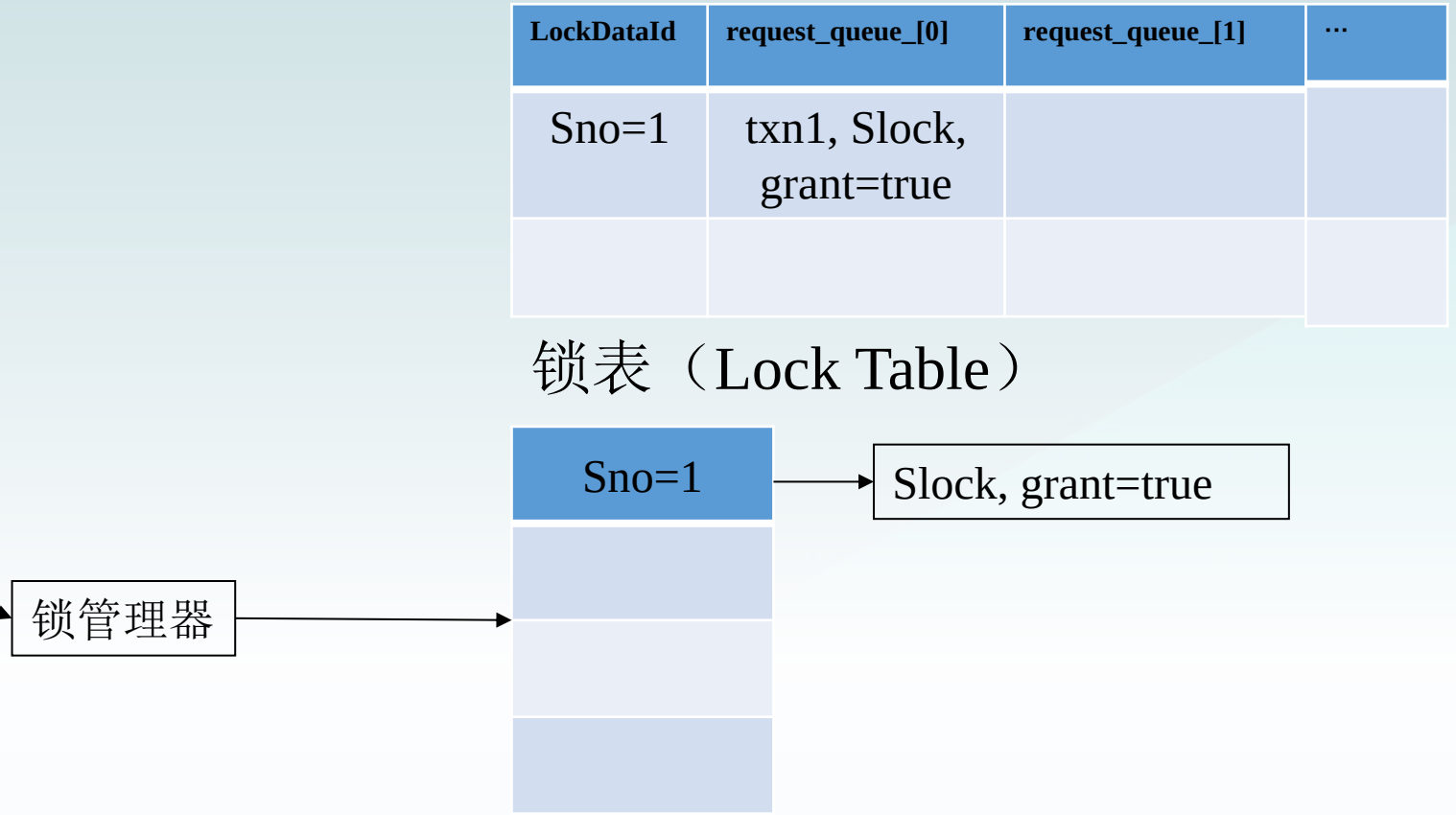
LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true		

锁表（Lock Table）

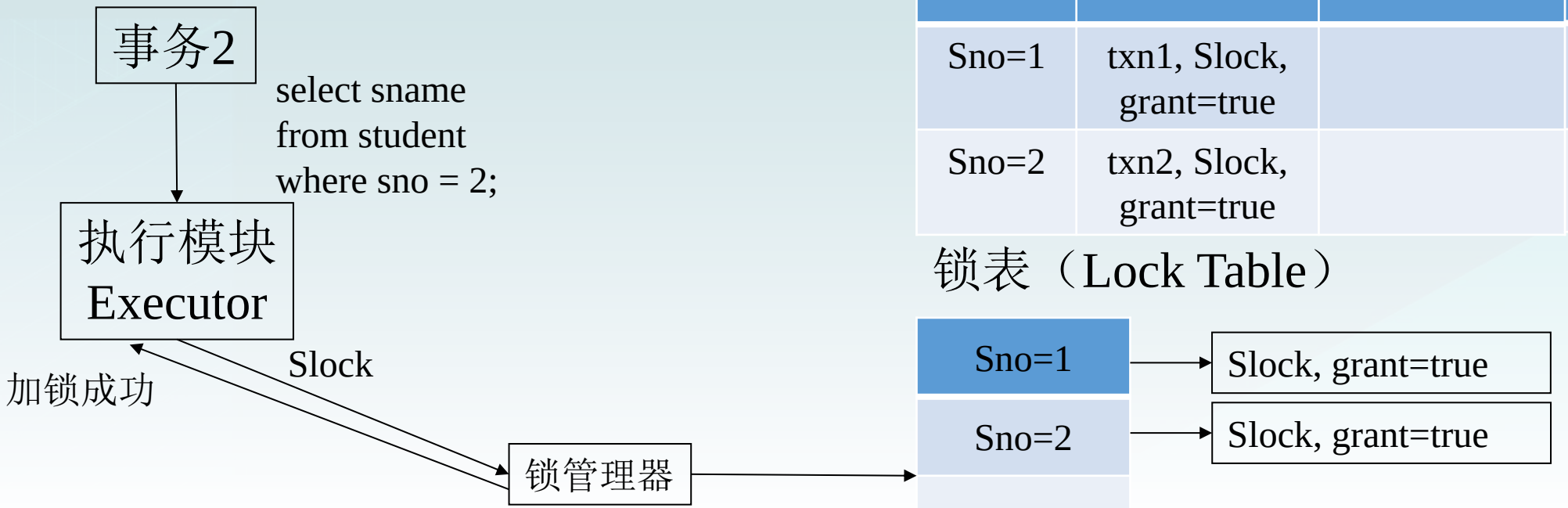


2PL


事务2
select sname
from student
where sno = 2;
执行模块
Executor
Slock


LockDataId
request_queue_[0]
request_queue_[1]
…
Sno=1
txn1, Slock,
grant=true
锁表（Lock Table）
Sno=1
Slock, grant=true
锁管理器

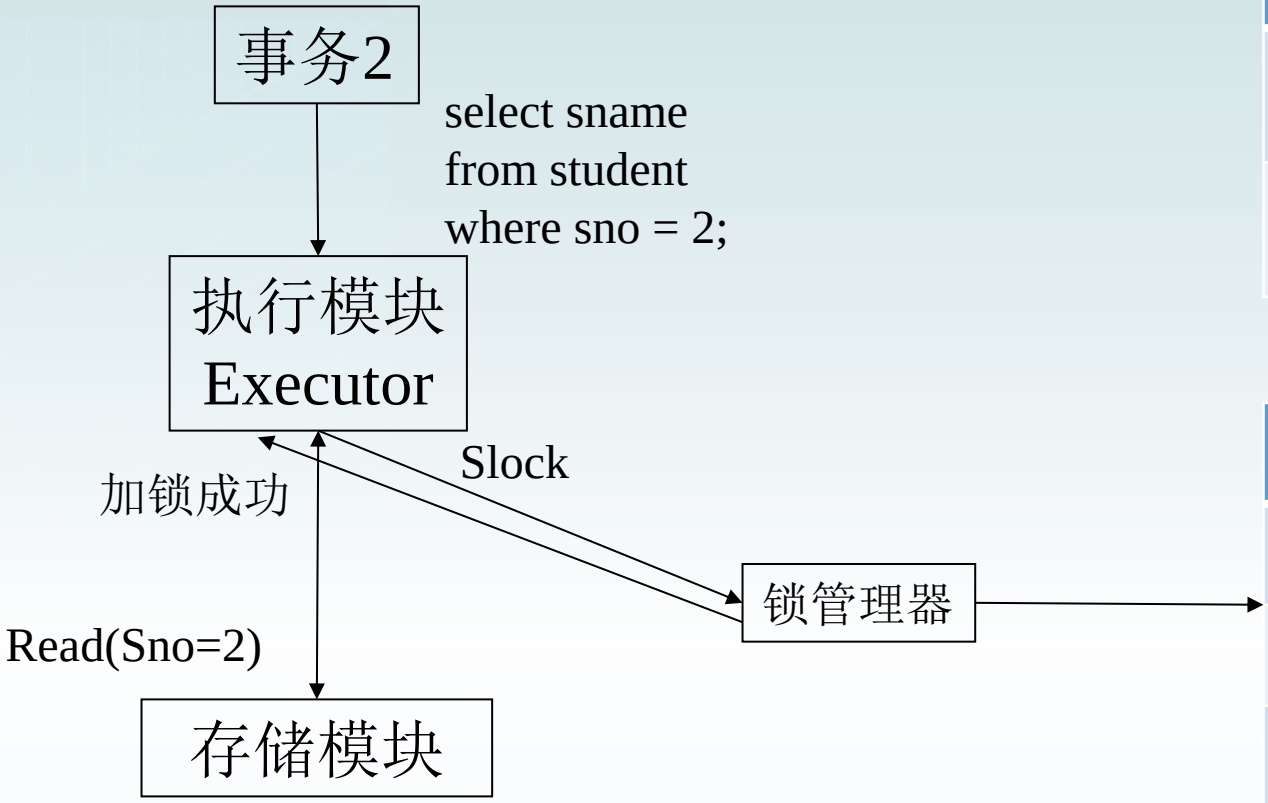
2PL



LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true		
Sno=2	txn2, Slock, grant=true		

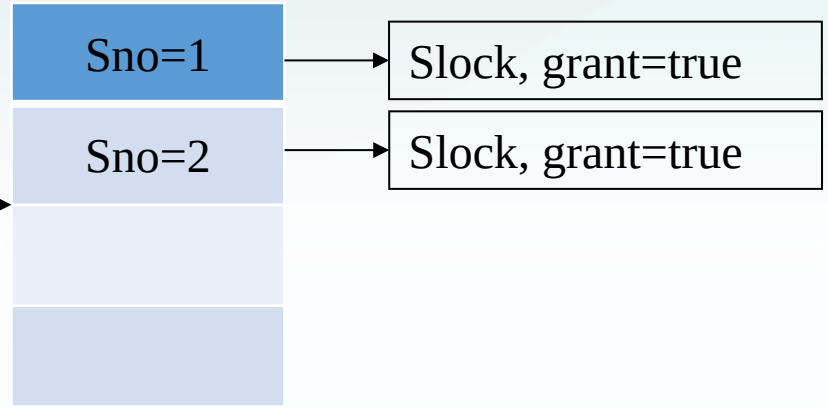
锁表（Lock Table）

2PL



LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true		
Sno=2	txn2, Slock, grant=true		

锁表（Lock Table）



2PL

```
select sname
from student
where sno = 1;
```

事务1

```
update student
set sname = ‘xiaoli’
where sno = 2;
```

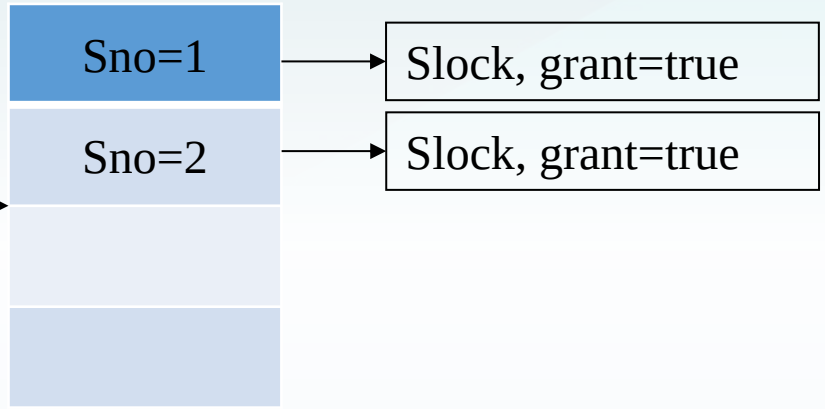
执行模块
Executor

Xlock

锁管理器

LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true		
Sno=2	txn2, Slock, grant=true		

锁表（Lock Table）



2PL

```
select sname
from student
where sno = 1;
```

事务1

```
update student
set sname = ‘xiaoli’
where sno = 2;
```

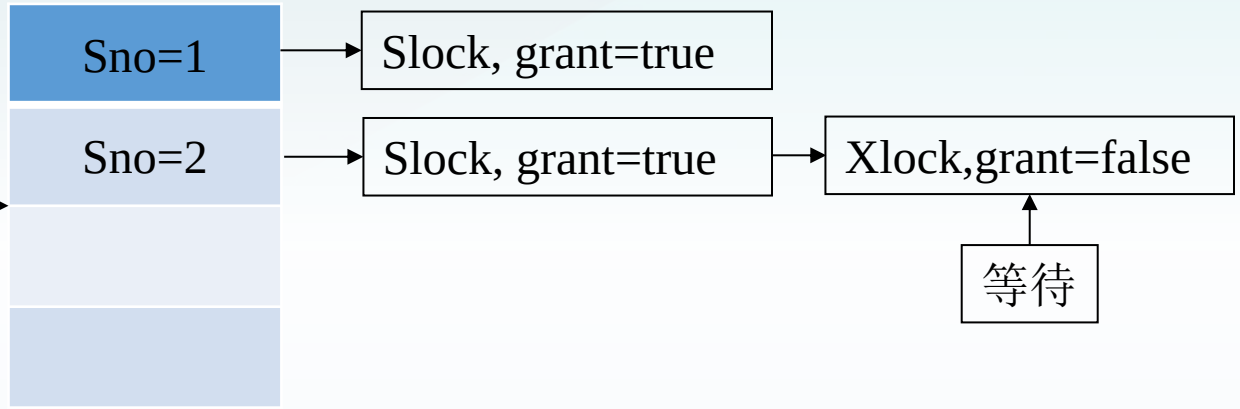
执行模块
Executor

Xlock

锁管理器

LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true		
Sno=2	txn2, Slock, grant=true	txn1, Xlock, grant=false	

锁表（Lock Table）



2PL

```
select sname
from student
where sno = 2;
```

事务2

```
update student
set sname = ‘xiaoli’
where sno = 1;
```

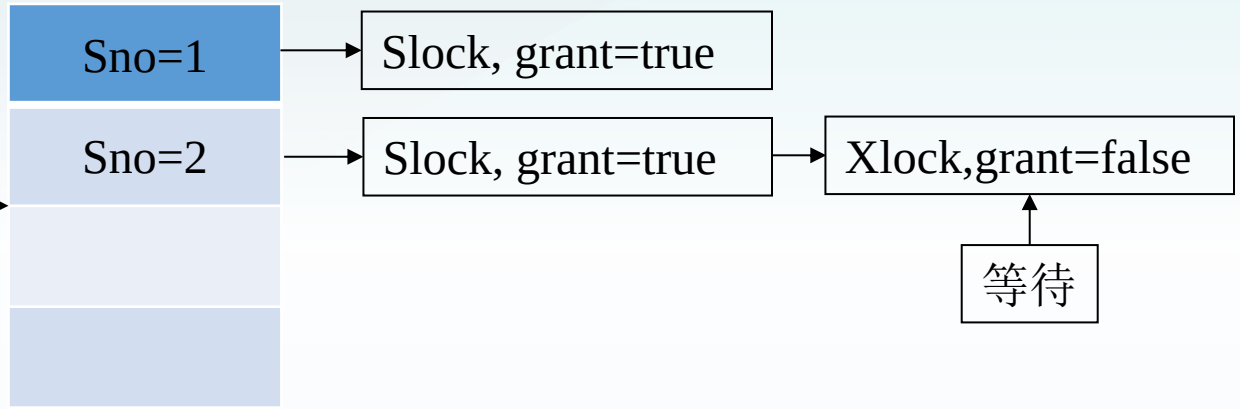
执行模块
Executor

Xlock

锁管理器

LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true		
Sno=2	txn2, Slock, grant=true	txn1, Xlock, grant=false	

锁表（Lock Table）



2PL

```
select sname
from student
where sno = 2;
```

事务1

```
update student
set sname = ‘xiaoli’
where sno = 1;
```

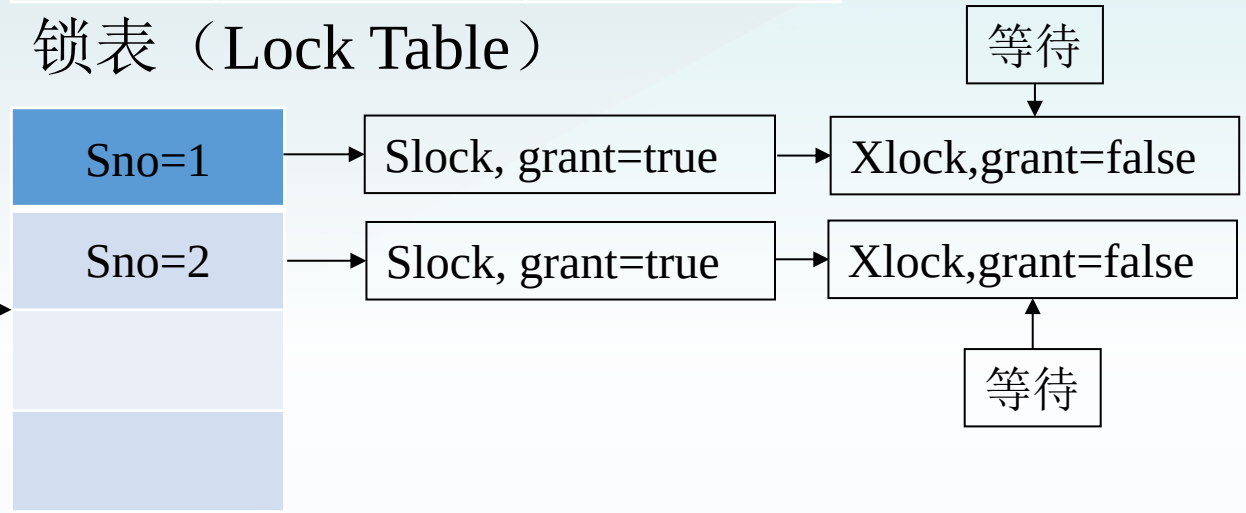
执行模块
Executor

Xlock

锁管理器

LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true	txn2, Xlock, grant=false	
Sno=2	txn2, Slock, grant=true	txn1, Xlock, grant=false	

锁表（Lock Table）



发生死锁！

2PL

```
select sname
from student
where sno = 2;
```

事务1

```
update student
set sname = ‘xiaoli’
where sno = 1;
```

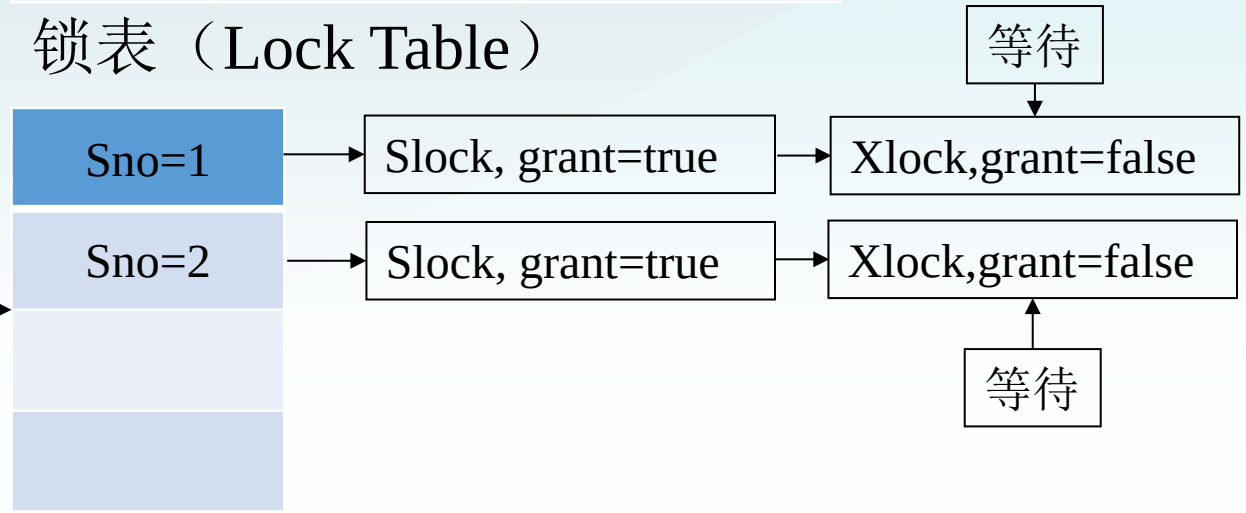
执行模块
Executor

Xlock

锁管理器

LockDataId	request_queue_[0]	request_queue_[1]	…
Sno=1	txn1, Slock, grant=true	txn2, Xlock, grant=false	
Sno=2	txn2, Slock, grant=true	txn1, Xlock, grant=false	

锁表（Lock Table）

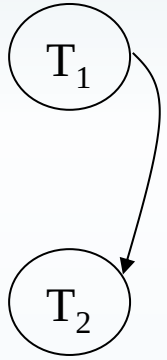


发生死锁！

两阶段封锁协议与冲突可串行化

- 基于事务优先图的冲突可串行化定义
  - 给定一组事务的调度SC_1，根据调度SC_1中事务之间的依赖关系，可以构建出等价的事务优先图$G(SC_1)$。其中，每个事务对应图中的一个顶点，图中的有向边为事务之间的依赖关系。SC_1是冲突可串行化的调度，当且仅当$G(SC_1)$中不存在环。

$SC_1=R_1(x)R_2(x)R_2(y)C_2W_1(x)W_1(y)C_1$



依赖1：$R_2(x)W_1(x)$
依赖2：$R_2(y)W_1(y)$

两阶段封锁协议与冲突可串行化

- 定理
  - 给定一组事务，如果每个事务的执行都遵守两阶段封锁协议，则该组事务的执行一定是冲突可串行化的。
- 背后的原理
  - 每个事务的执行都遵守两阶段封锁协议，则该调度对应的事务依赖图中一定不存在环。
  - 为什么？

两阶段封锁协议与冲突可串行化（续）

- 事务依赖的传递性
  - 给定事务Ti, Tj, Tk，如果Ti → Tj, Tj → Tk,则Ti → Tj
- 单向依赖
  - 给定任意两个事务Ti,Tj，Ti → Tj与Ti → Tj不能同时成立
- 死锁

事务T_1	事务T_2
Slock B	
R(B)=2	
	Slock A
	R(A)=2
Xlock A	
等待	Xlock A
等待	等待

遵守两段锁协议的事务可能发生死锁

两阶段封锁协议与冲突可串行化（续）

- 事务依赖的传递性
  - 给定事务Ti, Tj, Tk，如果Ti → Tj, Tj → Tk,则Ti → Tj
- 单向依赖
  - 给定任意
- 死锁

每个事务的执行都遵守两阶段封锁协议，事务之间要么只存在单向依赖，要么出现死锁

Slock B	
R(B)=2	
	Slock A
	R(A)=2
Xlock A	
等待	Xlock A
等待	等待

遵守两段锁协议的事务可能发生死锁

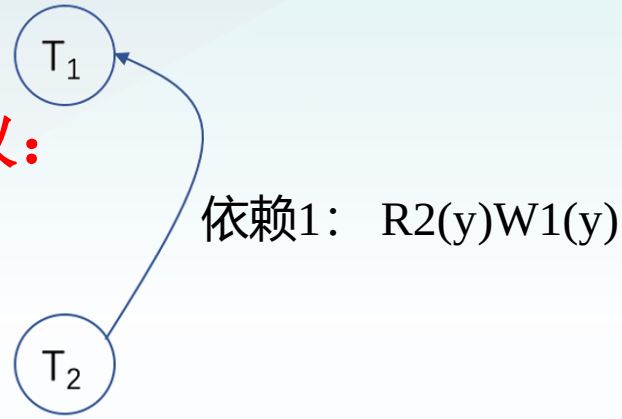
两阶段封锁协议与冲突可串行化（续）

- 给定T_1和T_2，其中T_1：$R_1(x)W_1(x)W_1(y)$；T_2：$R_2(x)R_2(y)$

遵循两阶段封锁协议的事务调度：　$SC_2=R_1(x)R_2(x)R_2(y)W_1(x)W_1(y)C_2C_1$

步骤	T1	T2
①	Slock x	
②	R(x)	
③		Slock x
④		R(x)
⑤	Xlock x	
⑥	等待	
⑦		Slock y
⑧		R(y)
⑨		Commit;
⑩		Unlock x, y
⑪	获得Xlock x	
⑫	W(x)	
⑬	Xlock y	
⑭	W(y)	
⑮	Commit;	
⑯	Unlock x, y	

- 遵循两阶段封锁协议：
  - 单向依赖



两阶段封锁协议与冲突可串行化（续）

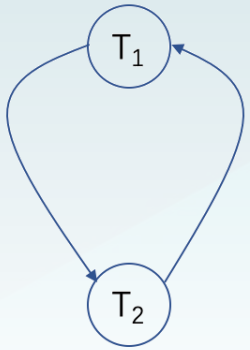
- 例：SC1=R1(x)R2(y)R1(y)R2(x)W1(x)W2(y)W1(y)W2(x)C1C2

遵循两阶段封锁协议的事务调度：

步骤	T1	T2
①	Slock x	
②	R(x)	
③		Slock y
④		R(y)
⑤	Slock y	
⑥	R(y)	
⑦		Slock x
⑧		R(x)
⑨	Xlock x	
⑩	等待	Xlock y
⑪		等待

- SC1的数据依赖图：

依赖1：R2(y)W1(y)
依赖2：W2(y)W1(y)
依赖3：R2(x)W1(x)



依赖1：R1(x)W2(x)
依赖2：W1(x)W2(x)
依赖3：R1(y)W2(y)

- 遵循两阶段封锁协议：
  - 发生了死锁
  - 没有数据依赖

两阶段封锁协议

- 事务遵守两段锁协议是可串行化调度的充分条件，而不是必要条件。
  - 若并发事务都遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的
  - 若并发事务的一个调度是可串行化的，不一定所有事务的执行都遵循两段锁协议

两阶段封锁协议存在的问题

- 问题1：死锁
- 问题2：级联回滚

步骤	T_1	T_2
①	Begin;	
②	Xlock A	
③	W_1(A=100);	Begin;
④		Slock A
⑤		等待
⑥		等待
⑦	Unlock A	
⑧		R_2(A): A=100
⑨		C_2
⑩	A_1	

不能提交。必须等到T_1提交之后T_2才能提交；并且，T_1回滚必然导致T_2回滚

两阶段封锁协议存在的问题的解决办法

- 死锁的解决办法
  - 死锁预防
  - 死锁避免
- 级联回滚的解决办法
  - 严格两阶段封锁协议（Strict 2PL）
  - 强严格两阶段封锁协议（Strong Strict 2PL）

12.5 活锁和死锁

- 封锁技术可以有效地解决并行操作的一致性问题，但也带来一些新的问题
  - 死锁
  - 活锁

12.5 活锁和死锁

12.5.1 活锁

12.5.2 死锁

12.5.1 活锁

- 事务T1封锁了数据R
- 事务T2又请求封锁R，于是T2等待。
- T3也请求封锁R，当T1释放了R上的封锁之后系统首先批准了T3的请求，T2仍然等待。
- T4又请求封锁R，当T3释放了R上的封锁之后系统又批准了T4的请求……
- T2有可能永远等待，这就是活锁的情形

活锁（续）

- 如何避免活锁：采用先来先服务的策略
  - 当多个事务请求封锁同一数据对象时
  - 按请求封锁的先后次序对这些事务排队
  - 该数据对象上的锁一旦释放，首先批准申请队列中第一个事务获得锁

12.5 活锁和死锁

12.5.1 活锁

12.5.2 死锁

12.5.2 死锁

T_1	T_2
Lock R_1	•
	•
	•
•	Lock R_2
•	•
•	•
Lock R_2	•
等待	
等待	
等待	Lock R_1
等待	等待
等待	等待
	•
	•
	•

(b) 死锁

- 事务T_1封锁了数据R_1
- T_2封锁了数据R_2
- T_1又请求封锁R_2，因T_2已封锁了R_2，于是T_1等待T_2释放R_2上的锁
- 接着T_2又申请封锁R_1，因T_1已封锁了R_1，T_2也只能等待T_1释放R_1上的锁
- 这样T_1在等待T_2，而T_2又在等待T_1，T_1和T_2两个事务永远不能结束，形成**死锁**

解决死锁的方法

两类方法

1. 死锁的预防
2. 死锁的诊断与解除

1. 死锁的预防

- 产生死锁的原因是两个或多个事务都已封锁了一些数据对象，然后又都请求对已为其他事务封锁的数据对象加锁，从而出现死等待。

- 预防死锁的发生就是要破坏产生死锁的条件

死锁的预防（续）

预防死锁的方法

（1）一次封锁法

（2）顺序封锁法

（1）一次封锁法

- 要求每个事务必须一次将所有要使用的数据全部加锁，否则就不能继续执行

T_1	T_2
Lock R_1	.
	.
	Lock R_2
.	.
.	.
.	.
Lock R_2	
等待 等待 等待	
.	Lock R_1
.	等待
	.
	.

T_1	T_2
Lock R_1 Lock R_2	.
	.
.	Lock R_2
.	等待
.	等待
ULock R_1	.
ULock R_2	
	Lock R_2
	Lock R_1
	.
	.

一次封锁法存在的问题

- 过早加锁，降低系统并发度
- 难于事先精确确定封锁对象
  - 数据库中数据是不断变化的，原来不要求封锁的数据，在执行过程中可能会变成封锁对象，所以很难事先精确地确定每个事务所要封锁的数据对象
  - 解决方法：将事务在执行过程中可能要封锁的数据对象全部加锁，这就进一步降低了并发度

（2）顺序封锁法

- 顺序封锁法是预先对数据对象规定一个封锁顺序，所有事务都按这个顺序实行封锁。
- 顺序封锁法存在的问题
  - 维护成本高
    - 数据库系统中可封锁的数据对象极多，并且随数据的插入、删除等操作而不断地变化，要维护这样的资源的封锁顺序非常困难，成本很高

顺序封锁法（续）

- 难于实现
  - 事务的封锁请求可以随着事务的执行而动态地决定，很难事先确定每一个事务要封锁哪些对象，因此也就很难按规定的顺序去施加封锁。

例：规定数据对象的封锁顺序为A,B,C,D,E。 事务T3起初要求封锁数据对象B,C,E，但当它 封锁了B,C后，才发现还需要封锁A，这样就破 坏了封锁顺序.

死锁的预防（续）

- 结论
  - 在操作系统中广为采用的预防死锁的策略并不太适合数据库的特点
  - 数据库管理系统在解决死锁的问题上更普遍采用的是诊断并解除死锁的方法

2. 死锁的诊断与解除

- 允许死锁发生
- 解除死锁
  - 由DBMS的并发控制子系统定期检测系统中是否存在死锁
  - 一旦检测到死锁，就要设法解除

2. 死锁的诊断与解除

- 死锁的诊断

  （1）超时法

  （2）等待图法

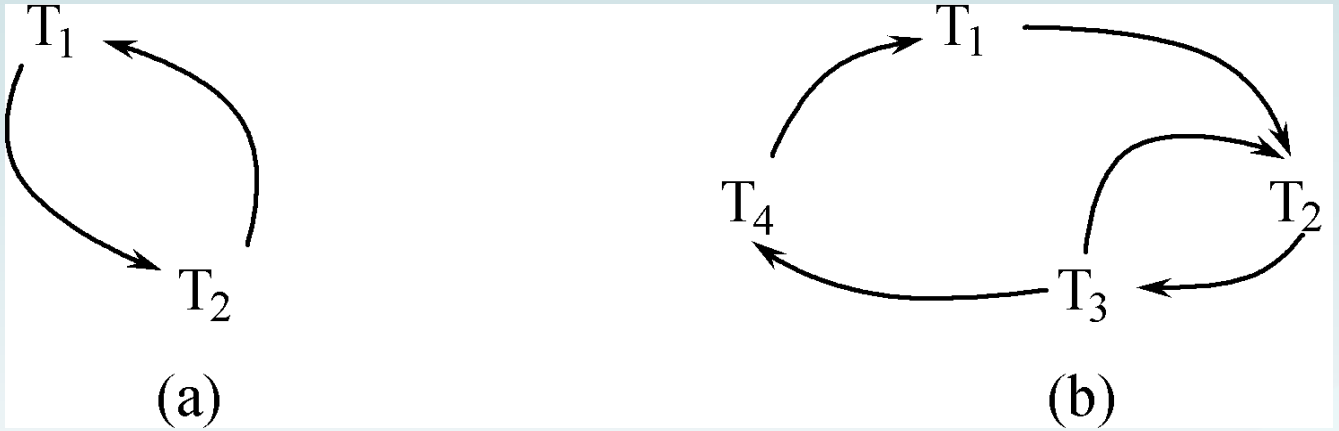
（1） 超时法

- 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) 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

等待图法（续）

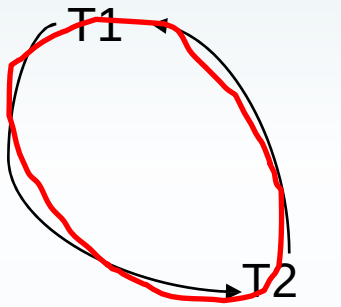


事务等待图

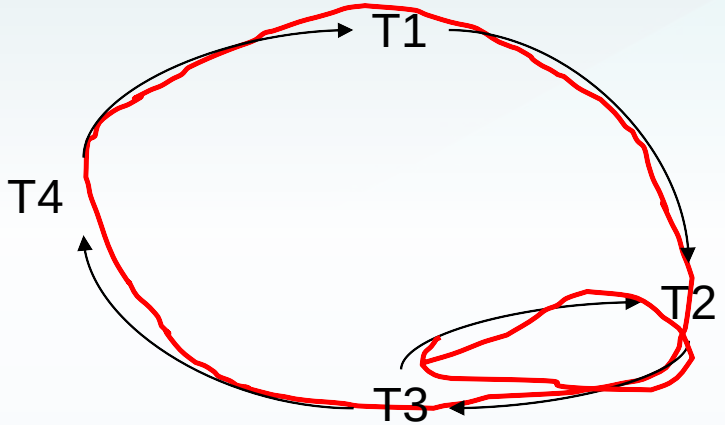
- 图(a)中，事务T_1等待T_2，T_2等待T_1，产生了死锁
- 图(b)中，事务T_1等待T_2，T_2等待T_3，T_3等待T_4，T_4又等待T_1，产生了死锁
- 图(b)中，事务T_3可能还等待T_2，在大回路中又有小的回路

等待图法（续）

- 并发控制子系统周期性地（比如每隔数秒）生成事务等待图，检测事务。如果发现图中存在回路，则表示系统中出现了死锁。



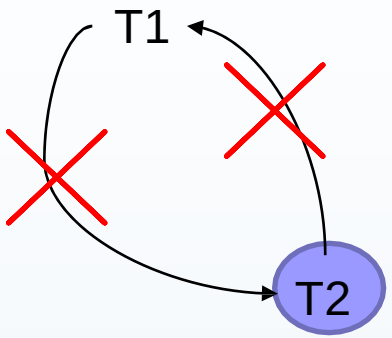
(a)



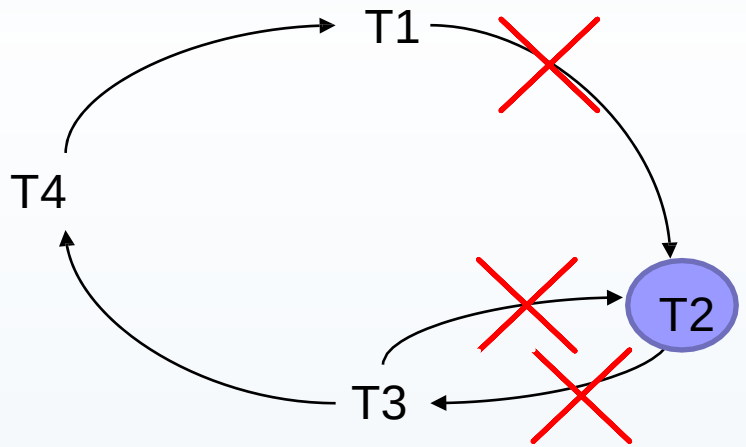
(b)

死锁的诊断与解除（续）

- 解除死锁
  - 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，使其它事务能继续运行下去。



(a)



(b)

11.1 两阶段封锁协议

- 11.1.1 两阶段封锁协议概念
- 11.1.2 严格与强严格两阶段封锁协议
- 11.1.3 死锁预防实现技术

如何处理级联回滚

- 严格两阶段封锁协议
  - 事务的执行除了遵守两阶段封锁协议之外，事务持有的排它锁必须在事务提交之后才能释放。
- 强严格两阶段封锁协议
  - 事务的执行除了遵守两阶段封锁协议之外，事务在提交之前不允许释放任何锁，即事务持有的所有锁必须在事务提交之后才能释放。

步骤	T_1	T_2
①	Begin;	
②	Xlock A	
③	W_1(A=100);	Begin;
④		Slock A
⑤		等待
⑥		等待
⑦	C_1	
⑧	Unlock A	
⑨		R_2(A): A=100
⑩		Unlock A
⑪		C_2

如何处理级联回滚

- 严格两阶段封锁协议
  - 事务的执行除了遵守两阶段封锁协议之外，事务持有的排它锁必须在事务提交之后才能释放。
- 强严格两阶段封锁协议
  - 事务的执行除了遵守两阶段封锁协议之外，事务在提交之前不允许释放任何锁，即事务持有的所有锁必须在事务提交之后才能释放。

步骤	T_1	T_2
①	Begin;	
②	Xlock A	
③	W_1(A=100);	Begin;
④		Slock A
⑤		等待
⑥		等待
⑦	C_1	
⑧	Unlock A	
⑨		R_2(A): A=100
⑩		C_2
⑪		Unlock A

11.1 两阶段封锁协议

- 11.1.1 两阶段封锁协议概念
- 11.1.2 严格与强严格两阶段封锁协议
- 11.1.3 死锁预防实现技术

基于死锁预防的两阶段封锁协议

- No-Wait（发生冲突就回滚）：
  - 当事务想要请求锁的时候，发现该数据项上的锁已经被其它事务持有时，回滚该事务。
- Wait-Die（优先级高的事务等待优先级低的事务）：
  - 如果请求锁的事务的优先级高于拥有锁的事务的优先级，那么请求锁的事务等待拥有锁的事务释放锁，否则回滚请求锁的事务。
  - 系统通常使用事务的开始时间戳来决定事务的优先级，事务的开始时间戳越小，优先级越高。
- Wound-Wait（优先级低的事务等待优先级高的事务）：
  - 如果请求锁的事务的优先级高于拥有锁的事务的优先级，那么拥有锁的事务回滚并释放占有的锁，否则请求锁的事务等待拥有锁的事务释放锁。

死锁预防（no-wait）

- No-Wait（发生冲突就回滚）：
  - 当事务想要请求锁的时候，发现该数据项上的锁已经被其它事务持有时，回滚该事务

步骤	T1	T2
1	Slock x	
2	R(x)	
3		Slock y
4		R(y)
5	Slock y	
6	R(y)	
7		Slock x
8		R(x)
9	Xlock x	
10	A_1	
11	Unlock x,y;	
12		Xlock y
13		$W_2(y,50)$
14		Xlock y
15		$W_2(x,150)$
16		C_2
17		Unlock x,y

死锁预防（wait-die）

- Wait-Die（优先级高的事务等待优先级低的事务）：

  - 如果请求锁的事务的优先级高于拥有锁的事务的优先级，那么请求锁的事务等待拥有锁的事务释放锁，否则回滚请求锁的事务。
  - 系统通常使用事务的开始时间戳来决定事务的优先级，事务的开始时间戳越小，优先级越高。

步骤	T1	T2
1	Slock x	
2	R(x)	
3		Slock y
4		R(y)
5	Slock y	
6	R(y)	
7		Slock x
8		R(x)
9	Xlock x	
10	等待//优先级高	
11		Xlock y
12		A_2//优先级低
13		Unlock x,y
14	$W_1(x,50)$	
15	Xlock y	
16	$W_1(x,150)$	
17	C_1	
18	Unlock x,y	

死锁预防（wound-wait）

- Wound-Wait（优先级低的事务等待优先级高的事务）：

  - 如果请求锁的事务的优先级高于拥有锁的事务的优先级，那么拥有锁的事务回滚并释放占有的锁，否则请求锁的事务等待拥有锁的事务释放锁。

步骤	T1	T2
1	Slock x	
2	R(x)	
3		Slock y
4		R(y)
5	Slock y	
6	R(y)	
7		Slock x
8		R(x)
9	Xlock x	
10	抢占//优先级高	
11		A_2//优先级低
12		Unlock x,y
13	W_1(x,50)	
14	Xlock y	
15	W_1(x,150)	
16	C_1	
17	Unlock x,y	

讨论

- 为什么no-wait/wait-die/wound-wait等策略可以保证不出现死锁？
  - 在死锁预防策略下，系统中只可能出现单向等待（优先级高的事务等待优先级低的事务，或优先级低的事务等待优先级高的事务）
- 当一个事务重启时，如何制定它的新的优先级？
  - 使用它的原来的时间戳作为它的新优先级
  - 为什么？

两阶段锁协议总结

- 如何定序
  - 事务在运行的过程中，一旦在某个数据项上出现冲突，则进行定序。
  - 给定两个并发事务T1，T2，假设T1和T2在数据项A上存在冲突，且T2的操作在前，T1的操作在后，则定序的规则：T2->T1,即并发事务T1、T2等价的可串行化顺序是T2->T1。
- 如何检验
  - 检验什么：是否有锁
  - 读操作，检查读取的数据项上是否存在排它锁
  - 写操作，检查要修改的数据项上是否存在排它锁和共享锁

第11章 并发控制

- 11.1 两阶段封锁协议
- 11.2 时间戳排序协议
- 11.3 乐观并发控制协议
- 11.4 三级封锁协议
- 11.5 多版本并发控制

基于时间戳的并发访问控制-TO

- 如何定序
  - 按事务的开始时间排序。TS（T）表示事务T的开始时间
  - Ti -> Tj if TS（Ti）< TS（Tj）
- 如何检验
  - 检验什么：冲突操作的顺序与开始时间序是否相同
    - 读写冲突：Ri(x)Wj(x) Ti -> Tj
    - 写读冲突：Wi(x)Rj(x) Ti -> Tj
    - 写写冲突：Wi(x)Wj(x) Ti -> Tj

基于时间戳的并发访问控制-TO

- 如何检验
- 检验什么：冲突操作
  - 读写冲突：Ri(x)Wj(x) Ti -> Tj
  - 写读冲突：Wi(x)Rj(x) Ti -> Tj
  - 写写冲突：Wi(x)Wj(x) Ti -> Tj
- 什么时候检验：事务执行的任何一个操作
- 检验不通过：如何做？回滚

基于时间戳的并发访问控制-TO

- 如何检验
- 检验什么：冲突操作
  - 读写冲突：Ri(x)Wj(x) Ti -> Tj
  - 写读冲突：Wi(x)Rj(x) Ti -> Tj
  - 写写冲突：Wi(x)Wj(x) Ti -> Tj
- 什么时候检验：事务执行的任何一个操作
- 检验不通过：如何做？回滚

步骤	T1	T2
①	**Begin;**	
②		**Begin;**
③		**R_2(A):A=100**
④		**W_2(A-50)**
⑤	**R_1(A)**	
⑥	**W_1(A-50)**	
⑦	**C_1;**	
⑧		**C_2;**

time ↓

基于时间戳的并发访问控制-TO

- 如何检验
- 检验什么：冲突操作
  - 读写冲突：Ri(x)Wj(x) Ti -> Tj
  - 写读冲突：Wi(x)Rj(x) Ti -> Tj
  - 写写冲突：Wi(x)Wj(x) Ti -> Tj
- 什么时候检验：事务执行的任何一个操作
- 检验不通过：如何做？回滚

步骤	T1	T2
①	Begin; TS(T1)=1	
②		Begin; TS(T2)=2
③		$R_2(A)$:A=100
④		$W_2(A-50)$
⑤	$R_1(A)$ 检查不通过，回滚	
⑥	A_1;	
⑦		C_2;

time ↓

基于时间戳的并发访问控制-TO

- 算法实践
- 数据结构
  - TS(T):每个事务开始时都会被赋予一个时间戳（单调递增）
  - RTS(x):元组x上最近读取该数据的事务时间戳
  - WTS(x):元组x上最近写入该数据的事务时间戳

元组	RTS	WTS
x	10000	10000
y	10000	10000

基于时间戳的并发访问控制-TO

- 算法实践
- 算法逻辑
  - 事务Ti开始：分配TS(Ti）
  - 检验：
    - 读操作Ri(x)

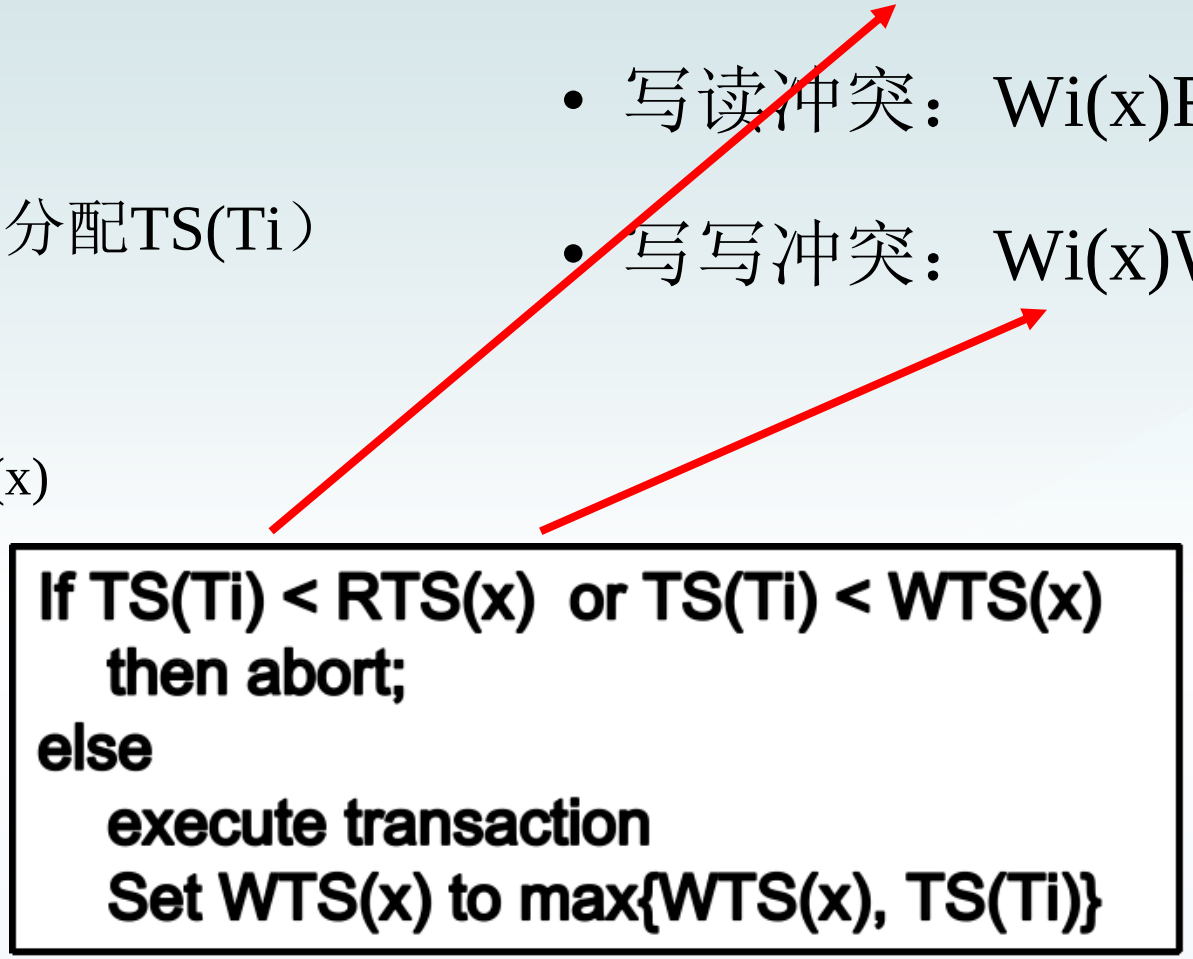
```
If TS(Ti) < WTS(x)
    then abort;
else
    execute transaction
    Set RTS(x) to max{RTS(x), TS(Ti)}
```

- 读写冲突：Ri(x)Wj(x) Ti->Tj
- 写读冲突：Wi(x)Rj(x) Ti->Tj
- 写写冲突：Wi(x)Wj(x) Ti->Tj

基于时间戳的并发访问控制-TO

- 算法实践
- 算法逻辑
  - 事务Ti开始：分配TS(Ti）
  - 检验：
    - 写操作Wi(x)

- 读写冲突： Ri(x)Wj(x) Ti->Tj
- 写读冲突： Wi(x)Rj(x) Ti->Tj
- 写写冲突： Wi(x)Wj(x) Ti->Tj



```
If TS(Ti) < RTS(x)  or TS(Ti) < WTS(x)
    then abort;
else
    execute transaction
    Set WTS(x) to max{WTS(x), TS(Ti)}
```

基于时间戳的并发访问控制-TO

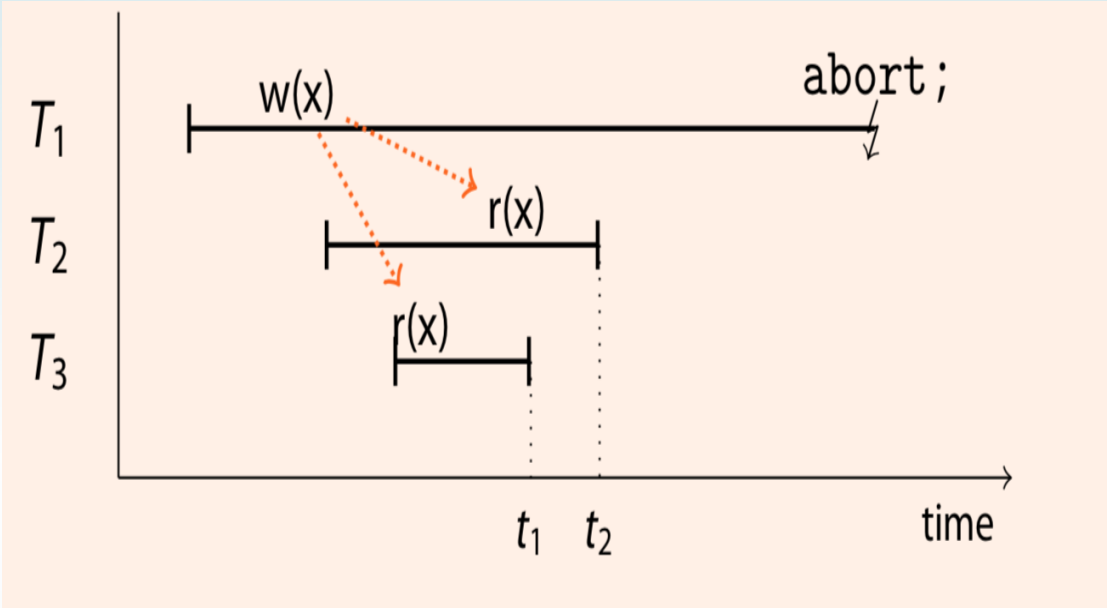
- 托马斯写是TO的一条特殊规则
  - 忽略过时的写：
    - 如果TS(T)< RTS(X)，则事务T被中止并回滚，并且操作被拒绝。
    - 如果TS(T)< WTS(X)，则不执行事务的写数据操作并继续处理，即忽略掉当前事务T的写入。
    - 如果条件1和条件2都不发生，则允许通过事务T正常执行写操作并将WTS(X)设置为TS(T)。

步骤	T1	T2
①	Begin; TS(T1)=1	
②		Begin; TS(T2)=2
③	$R_1(A)$	
④		$R_2(B)$
⑤		$W_2(C, B)$
⑥	$W_1(C, A)$ 过时版本，忽略	
⑦		C_2;
⑧	C_1;	

time ↓

TO存在的问题：级联回滚

如何处理级联回滚



解决的思路

（1）每个事务需要维护“依赖列表”

（2）依赖列表中的事务提交之后，才能允许提交

TO存在的问题：级联回滚

- 另一种思路“预写操作”：事务写操作时不真正写入数据，而是将要写的数据暂存下来，在事务提交时再写入数据库。
  - 当一个更新的读事务读取数据时，可能会发现存在一个时间戳更旧的预写事务还未提交，为了满足时间戳顺序，在这种情况下读事务必须要等待。

TO存在的问题：级联回滚

- 为此数据项X上需要额外维护五个元信息：
  - Min-PTS，代表当前数据项上所有预写操作的最小时间戳，用来判断读操作是否需要等待；
  - Min-RTS，代表当前数据项上所有预读操作的最小时间戳，用于判断写入是否要等待。
  - 三个队列来分别记录三类处于等待状态的事务：R-reqs，代表数据项上等待的读操作；P-reqs，代表当前数据项上的预写操作；W-reqs，代表当前数据项上等待的提交操作。

预写操作

- 第②步时，事务T1并不会直接将x的新值1写入数据库，而是将事务T1存入P-reqs(x)，代表事务T1会修改数据项x。
- 第③④步时，事务T2和T3会发现P-reqs(x)存在一个更小的事务T1，因此T2和T3会陷入等待。
- 当⑤事务T1回滚之后，事务T2和T3重新读取数据项x，正确提交（⑥⑦⑧）。

步骤	T1	T2	T3
①	Begin; TS(T1)=1		
②	$W_1(X,1)$	Begin; TS(T2)=2	
③		$R_2(X)$ 等待	Begin; TS(T3)=3
④			$R_3(X)$ 等待
⑤	A_1;		
⑥		$R_2(X,0)$	
⑦		C_2;	$R_3(X,0)$
⑧			C_3;

TO协议总结

- 如何定序
  - 事务开始时会被分配一个时间戳，事务之间根据时间戳的大小关系来排序，时间戳更小的排在前面。
- 如何检验
  - 事务执行过程中，一旦在某个数据项上出现冲突，则需要进行检验
  - 检验什么：如果该操作是读操作，则检查当前事务的时间戳TS(T)是否小于W-TS(X)，若小于，则代表当前事务的实际执行顺序与时间戳顺序相反，回滚当前事务；如果该操作是写操作，则检查当前事务的时间戳TS(T)是否小于MAX(WTS(X),RTS(X)），若小于，则代表当前事务的实际执行顺序与时间戳顺序相反，回滚当前事务。

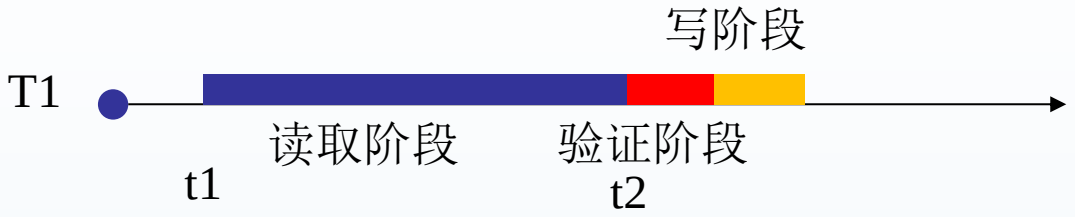
第11章 并发控制

- 11.1 两阶段封锁协议
- 11.2 时间戳排序协议
- 11.3 乐观并发控制协议
- 11.4 三级封锁协议
- 11.5 多版本并发控制

乐观并发访问控制（OCC）

- 主要思想：事务的生命周期被划分为三个阶段
  - 读取阶段：执行事务，但不进行写操作。**更新的数据维护缓存在事务的私有空间。**
  - 验证阶段：检测事务是否满足可串行化隔离级别。**如何检查？**
  - 写阶段：**将事务私有空间中的更新数据写入数据库**
- 注意：设置临界区，在同一时刻，只允许一个事务进入验证和写阶段。

（原子操作）



乐观并发访问控制

- 乐观并发访问控制（Optimistic Concurrency Control，OCC）
  - 读取阶段：所有读取的数据会拷贝到本地空间，所有写也只记录到本地空间
  - 验证阶段：事务执行commit的时候，DBMS会先检查该事务是否和其他事务有冲突，验证阶段需要对事务修改的数据进行加锁
  - 写阶段：把本地空间的修改apply到DBMS，让其他事务可见，最后释放验证阶段加的锁

乐观并发访问控制

- 如何定序
  - 按事务的**验证时间**排序：如果Ti的验证时间早于Tj的验证时间，那么Ti -> Tj
- 如何检验
  - 检验什么：冲突操作
    - 读写冲突：Ri(x)Wj(x) Ti -> Tj
    - 写读冲突：Wi(x)Rj(x) Ti -> Tj
    - 写写冲突：Wi(x)Wj(x) Ti -> Tj
  - 什么时候检验：**验证阶段**
  - 检验不通过：**回滚**

乐观并发访问控制

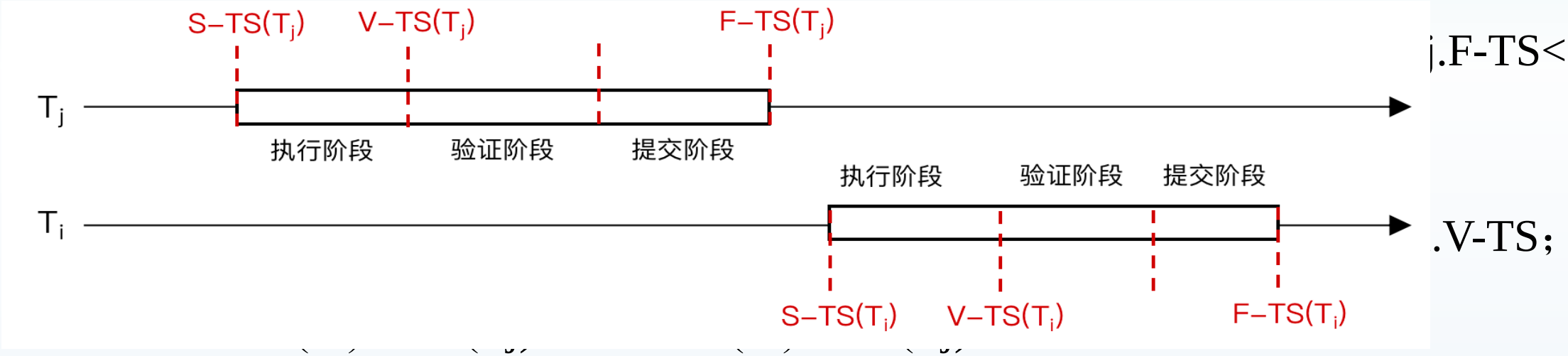
- 算法实践
  - 数据结构
    - 每个事务T：维护一个读写集合。读集T.RS为事务T所读过的数据项的集合；写集T.WS为事务T所写过的数据项的集合
    - T.ID为事务T的ID，全局唯一
    - T.S-TS事务的开始时间戳，即进入执行阶段的时间
    - T.V-TS事务的验证时间戳，即进入验证阶段的时间
    - T.F-TS事务的提交时间戳

乐观控制法（OCC）

- 在进行读取阶段，每个事务Ti维护两个集合
  - 读集（Read Set）：**Ti.RS**
  - 写集（Write Set）：**Ti.WS**
- 如何进行事务T验证阶段的检查？
  - **检查的对象**：事务Ti开始时尚未提交、事务Ti进入验证时完成提交的事务Tj
  - **检查的条件**： $Ti.RS \cap Tj.WS = \emptyset$

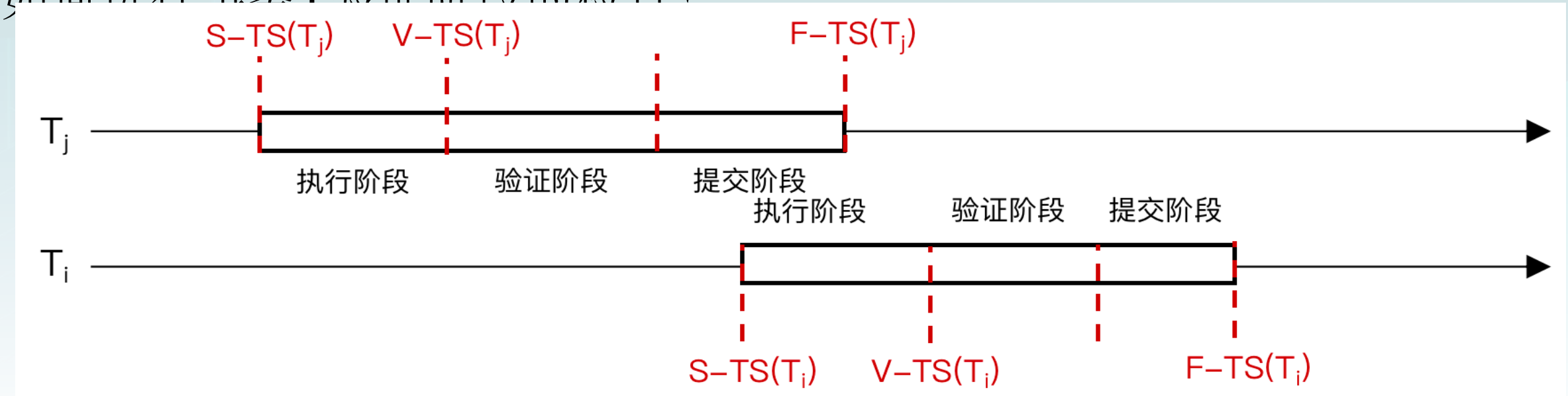
乐观控制法（OCC）

- 如何进行事务T验证阶段的检查？
  - **检查的对象**：事务Ti开始时尚未提交、事务Ti进入验证时完成提交的事务Tj
  - 具体的检查条件有三：
    - **若事务Tj在事务Ti开始时已经完成提交阶段，即Tj.F-TS< Ti.S-TS；此时Tj与Ti不存在冲突，无需进行检查**



乐观控制法（OCC）

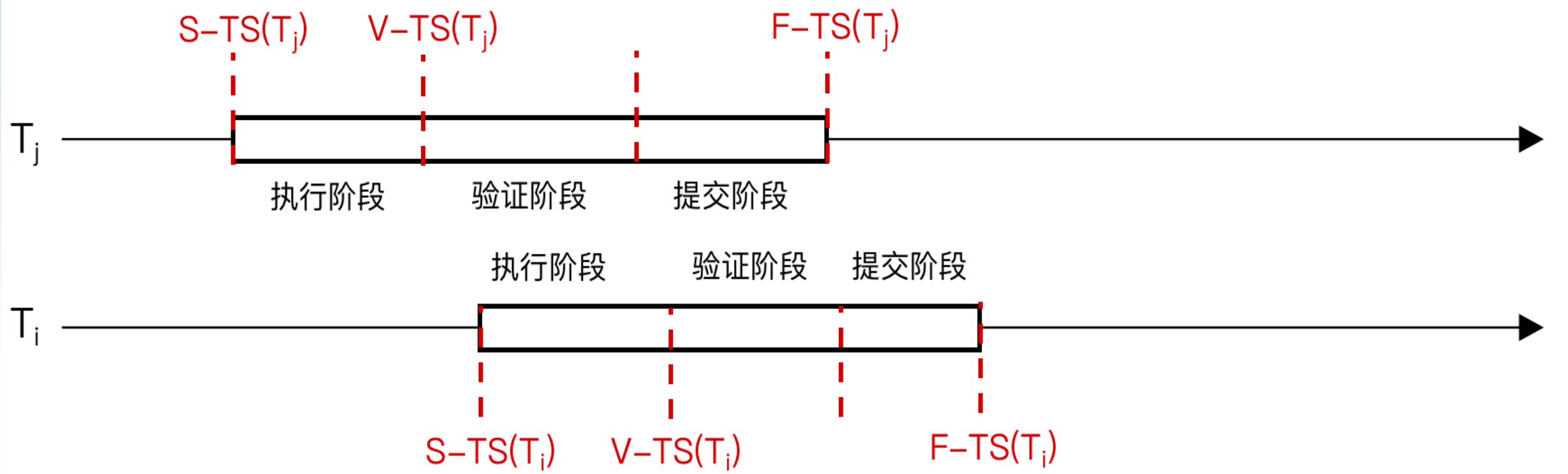
- 如何进行事务T验证阶段的检查？



- **若事务Tj在事务Ti进入验证阶段之前已经完成提交阶段，即Ti.S-TS<Tj.F-TS<Ti.V-TS；则需要满足Ti.RS∩Tj.WS = Ø；**
- 若事务Tj在事务Ti之前完成执行阶段，即Tj.S-TS<Ti.S-TS<Tj.V-TS<Ti.V-TS；需要满足RS(Ti)∩WS(Tj) = Ø且WS(Ti)∩WS(Tj) = Ø

乐观控制法（OCC）

- 如何进行事务T验证阶段的检查？



- **若事务Tj在事务Ti之前完成执行阶段，即Tj.S-TS<Ti.S-TS<Tj.V-TS<Ti.V-TS；**

  需要满足RS(Ti)∩WS(Tj) = Ø且WS(Ti)∩WS(Tj) = Ø

表 11.12　乐观并发控制协议示例

步骤	T_1	T_2	事务读写集
1	Begin $S\text{-}TS(T_1)=1$		$RS(T_1)\text{-}\Phi$; $WS(T_1)\text{-}\Phi$ $RS(T_2)\text{-}\Phi$; $WS(T_2)\text{-}\Phi$
2		Begin $S\text{-}TS(T_2)=2$	$RS(T_1)\text{-}\Phi$; $WS(T_1)\text{-}\Phi$ $RS(T_2)\text{-}\Phi$; $WS(T_2)\text{-}\Phi$
3	$R_1(A, 25)$ $R_1(B, 5)$		$RS(T_1)\text{-}\{A, B\}$; $WS(T_1)\text{-}\Phi$ $RS(T_2)\text{-}\Phi$; $WS(T_2)\text{-}\Phi$
4		$R_2(B, 5)$ $R_2(A, 25)$	$RS(T_1)\text{-}\{A, B\}$; $WS(T_1)\text{-}\Phi$ $RS(T_2)\text{-}\{A, B\}$; $WS(T_2)\text{-}\Phi$

步骤	T_1	T_2	事务读写集
5	$W_1(A, 20)$ $W_1(B, 10)$		$RS(T_1)$-{A,B}; $WS(T_1)$-{A,B} $RS(T_2)$-{A,B}; $WS(T_2)$-Φ
6	Validate V-$TS(T_1)=3$		$RS(T_1)$-{A,B}; $WS(T_1)$-{A,B} $RS(T_2)$-{A,B}; $WS(T_2)$-Φ
7	Commit F-$TS(T_1)=4$		$RS(T_1)$-{A,B}; $WS(T_1)$-{A,B} $RS(T_2)$-{A,B}; $WS(T_2)$-Φ
8		$W_2(B, 0)$ $W_2(A, 30)$	$RS(T_1)$-{A,B}; $WS(T_1)$-{A,B} $RS(T_2)$-{A,B}; $WS(T_2)$-{A,B}
9		Validate V-$TS(T_2)=5$	$RS(T_1)$-{A,B}; $WS(T_1)$-{A,B} $RS(T_2)$-{A,B}; $WS(T_2)$-{A,B}
10		Abort	

OCC协议总结

- 如何定序
  - 事务在验证阶段会被分配一个时间戳，事务之间根据时间戳的大小关系来排序，时间戳更小的排在前面。
- 如何检验
  - 检验什么：事务的验证阶段中，检查事务的读集是否与其并发事务的写集存在交集，若存在则回滚当前事务。

内容提要

- 可串行化下的并发控制
- 弱隔离级别下的并发控制

现有数据库系统的隔离级别

- 现有数据库系统默认的隔离级别
  - 现有的数据库系统默认隔离级别为弱隔离级别如读已提交。
  - 这是因为弱隔离级别拥有更高的性能，而且部分互联网应用并不存在隔离级别要求规避的异常。

编号	数据库管理系统	默认隔离级别	最高支持的隔离级别
1	Oracle 11g	读已提交	快照隔离
2	PostgreSQL 9.2.2	读已提交	可串行化
3	MySQL 5.6	可重复读	可串行化
4	Kingbase ESV8	读已提交	可串行化
5	openGauss 3.0	读已提交	可串行化
6	SQL Server 2012	读已提交	可串行化

第11章 并发控制

- 11.1 两阶段封锁协议
- 11.2 时间戳排序协议
- 11.3 乐观并发控制协议
- 11.4 三级封锁协议
- 11.5 多版本并发控制

保持数据一致性的常用封锁协议

- 三级封锁协议

  1.一级封锁协议

  2.二级封锁协议

  3.三级封锁协议

1. 一级封锁协议

- 一级封锁协议
  - 事务T在修改数据R之前必须先对其加X锁，直到事务结束才释放。
    - 正常结束（COMMIT）
    - 非正常结束（ROLLBACK）
- 一级封锁协议可防止丢失修改，并保证事务T是可恢复的。
- 在一级封锁协议中，如果仅仅是读数据不对其进行修改，是不需要加锁的，所以它不能保证可重复读和不读“脏”数据。

使用封锁机制解决丢失修改问题

例：

T_1	T_2
① R(A)=16	
	R(A)=16
②	
③ A←A-1	
W(A)=15	
	A←A-1
④	W(A)=15

	T_1	T_2
①	Xlock A	
②	R(A)=16	
		Xlock A
③	A←A-1	等待
	W(A)=15	等待
	Commit	等待
	Unlock A	等待
④		获得Xlock A
		R(A)=15
		A←A-1
⑤		W(A)=14
		Commit
		Unlock A

没有丢失修改

- 事务T_1在读A进行修改之前先对A加X锁
- 当T_2再请求对A加X锁时被拒绝
- T_2只能等待T_1释放A上的锁后获得对A的X锁
- 这时T_2读到的A已经是T_1更新过的值15
- T_2按此新的A值进行运算，并将结果值A=14写回到磁盘。避免了丢失T_1的更新。

一级封锁协议

- 在一级封锁协议中，如果仅仅是读数据不对其进行修改，是不需要加锁的，所以它不能保证可重复读和不读“脏”数据。

使用一级封锁协议不能解决的问题

T_1	T_2
① R(A)=50 R(B)=100 求和=150	
②	R(B)=100 B←B*2 W(B)=200
③ R(A)=50 R(B)=200 求和=250 验算不对	

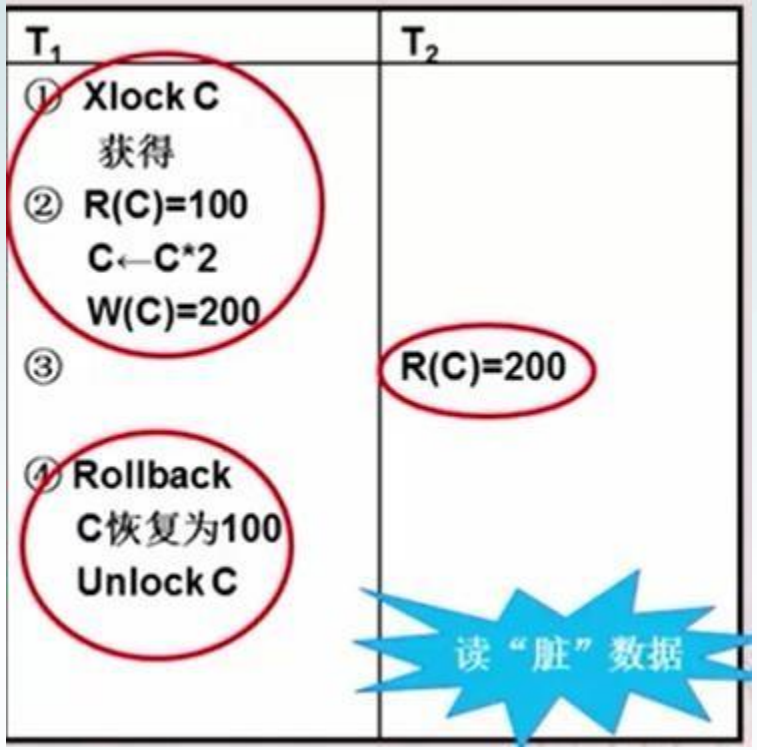
不可重复读

T_1	T_2
① R(A)=50 R(B)=100 求和=150	
②	XlockB 获得 R(B)=100 B←B*2 W(B)=200 Commit Unlock B
③ R(A)=50 R(B)=200 求和=250 验算不对	

使用一级封锁协议不能解决的问题

T_1	T_2
① R(C)=100	
C←C*2	
W(C)=200	
②	R(C)=200
③	
ROLLBACK	
C恢复为100	

读“脏”数据



2. 二级封锁协议

- 二级封锁协议
  - 一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，读完后即可释放S锁。

- 二级封锁协议可以防止丢失修改和读“脏”数据。

- 在二级封锁协议中，由于读完数据后即可释放S锁，所以它不能保证可重复读。

使用封锁机制解决读“脏”数据问题

例

T_1	T_2
① R(C)=100 C←C*2 W(C)=200 ② ③ ROLLBACK C恢复为100	R(C)=200

T_1	T_2
① Xlock C	
R(C)=100	
C←C*2	
W(C)=200	
②	Slock C
	等待
③ROLLBACK	等待
(C恢复为100)	等待
Unlock C	等待
④	获得Slock C
	R(C)=100
⑤	Commit C
	Unlock C

不读“脏”数据

- 事务T_1在对C进行修改之前，先对C加X锁，修改其值后写回磁盘
- T_2请求在C上加S锁，因T_1已在C上加了X锁，T_2只能等待
- T_1因某种原因被撤销，C恢复为原值100
- T_1释放C上的X锁后T_2获得C上的S锁，读C=100。避免了T_2读“脏”数据

二级封锁协议

- 在二级封锁协议中，由于读完数据后即可释放S锁，所以它不能保证可重复读。

使用二级封锁协议不能解决的问题

T_1	T_2
① R(A)=50	
R(B)=100	
求和=150	
②	R(B)=100
	B←B*2
	W(B)=200
③ R(A)=50	
R(B)=200	
求和=250	
(验算不对)	

T_1	T_2
① Sclock A	
获得	
读A=50	
Unlock A	
② Sclock B	
获得	
③	Xlock B
④ 读B=100	等待
Unlock B	等待
求和=150	
⑤	获得
	读B=100
	B←B*2
	写回B=200
	Commit
	Unlock B

T_1(续)	T_2
⑥ Sclock A	
获得	
读A=50	
Unlock A	
Sclock B	
获得	
读B=200	
Unlock B	
求和=250	
(验算不对)	

不可重复读

3. 三级封锁协议

- 三级封锁协议
  - 一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，直到事务结束才释放。

- 三级封锁协议可防止丢失修改、读脏数据和不可重复读。

使用二级封锁协议不能解决的问题

T_1	T_2
① R(A)=50	
R(B)=100	
求和=150	
②	R(B)=100
	B←B*2
	W(B)=200
③ R(A)=50	
R(B)=200	
求和=250	
(验算不对)	

T_1	T_2
① Sclock A	
获得	
读A=50	
Unlock A	
② Sclock B	
获得	
③	Xlock B
④ 读B=100	等待
Unlock B	等待
求和=150	
⑤	获得
	读B=100
	B←B*2
	写回B=200
	Commit
	Unlock B

T_1(续)	T_2
⑥ Sclock A	
获得	
读A=50	
Unlock A	
Sclock B	
获得	
读B=200	
Unlock B	
求和=250	
(验算不对)	

不可重复读

使用封锁机制解决不可重复读问题

T_1	T_2
① Slock A	
Slock B	
R(A)=50	
R(B)=100	
求和=150	
②	Xlock B
	等待
③ R(A)=50	等待
R(B)=100	等待
求和=150	等待
Commit	等待
Unlock A	等待
Unlock B	等待
④	获得XlockB
	R(B)=100
	B←B*2
⑤	W(B)=200
	Commit
	Unlock B

可重复读

- 事务T_1在读A，B之前，先对A，B加S锁
- 其他事务只能再对A，B加S锁，而不能加X锁，即其他事务只能读A，B，而不能修改
- 当T_2为修改B而申请对B的X锁时被拒绝只能等待T_1释放B上的锁
- T_1为验算再读A，B，这时读出的B仍是100，求和结果仍为150，即可重复读
- T_1结束才释放A，B上的S锁。T_2才获得对B的X锁

4．封锁协议小结

- 三级协议的主要区别
  - 什么操作需要申请封锁以及何时释放锁（即持锁时间）
- 不同的封锁协议使事务达到的一致性级别不同
  - 封锁协议级别越高，一致性程度越高

	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”数据	可重复读
一级封锁协议		√			√		
二级封锁协议		√	√		√	√	
三级封锁协议		√		√	√	√	√

表12.1 不同级别的封锁协议和一致性保证

课堂测试

1.如果事务T获得了数据项Q上的排他锁,则T对Q（）

A.只能读不能写　　B.只能写不能读

C.即可读又可写　　D.不能写也不能读

三级封锁协议

- 三级封锁协议

（1）一级封锁协议

（2）二级封锁协议

（3）三级封锁协议

（1）一级封锁协议

- 一级封锁协议
  - 事务T在修改数据A之前必须先对其加X锁，直到事务结束才释放
    - 正常结束（COMMIT）
    - 非正常结束（ROLLBACK）
  - 只有长写锁
- 一级封锁协议可防止脏写，并保证事务T是可恢复的。
- 在一级封锁协议中，如果仅仅是读数据不对其进行修改，是不需要加锁的，所以它不能保证可重复读和不读“脏”数据。

使用封锁机制解决脏写问题

例：

步骤	T1	T2
①	Begin;	
②		Begin;
③	R(A): A = 100;	
④		R(A): A = 100;
⑤	Xlock A	
⑥	W(A-50)	
⑦		Xlock A
⑧	Commit;	等待
⑨	Unlock A	
⑩		获得Xlock A
⑪		W(A-50)
⑫		Commit;
⑬		Unlock A

- 脏写的异常模式
  - ….W1(A)…W2(A)…..(C1…C2任何顺序)
- 加长写锁
  - 一旦T1拿到了写锁，T2必须等待T1释放
  - 模式….W1(A)…W2(A)…..C1不可能出现
  - 因为T1提交之后才会释放锁，之后T2才会拿到写锁
- 消除了脏写的异常模式
  - ….W1(A)…W2(A)…..(C1…C2任何顺序)

（2）二级封锁协议

- 二级封锁协议
  - 一级封锁协议加上事务T在读取数据A之前必须先对其加S锁，读完后即可释放S锁。
  - 长写锁：一级封锁协议
  - 段读锁：数据A读之前加读锁，读完后即可释放S锁
- 二级封锁协议可以防止脏写和读“脏”数据。
- 在二级封锁协议中，由于读完数据后即可释放S锁，所以它不能保证可重复读。

使用封锁机制解决读“脏”数据问题

例：

步骤	T1	T2
①	Begin;	
②		Begin;
③	Slock A	
④	R(A): A = 50;	
⑤	Xlock A	
⑥	W(A+50)	
⑦		Slock A
⑧		等待
⑨	Abort;	等待
⑩	Unlock A	
⑪		获得Slock A
⑫		R(A) = 50;

- 事务T_1在读取数据项A之前，先对A加S锁，读取完成之后释放S锁；
- 事务T_1在对数据项x进行写操作之前，先对A加X锁，修改其值为100；
- T_2请求在A上加S锁，因T_1已在A上加了X锁，T_2只能等待；
- T_1因某种原因被撤销，A恢复为原值50；
- T_1释放A上的X锁后T_2获得A上的S锁，读x=50，避免了T_2读“脏”数据。
- 不读脏数据，消除了以下异常模式
  - …$W_1(x)$…$R_2(x)$...A_1

（3）三级封锁协议

- 三级封锁协议
  - 一级封锁协议加上事务T在读取数据R之前必须先对其加S锁，直到事务结束才释放。
  - 长读锁
  - 长写锁
- 三级封锁协议可防止脏写、读脏数据和不可重复读。

使用封锁机制解决不可重复读问题

例：

步骤	T1	T2
①	Begin;	
②		Begin;
③	Slock A	
④	R(A)：A = 100;	
⑤		Xlock A
⑥		等待
⑦	R(A)：A = 100;	等待
⑧	Commit;	等待
	Unlock A	
⑨		获得Xlock A
⑩		R(A)：A = 0;
⑪		Commit;

可重复读

- 事务T1在读数据项A之前，先对A加S锁；
- 其他事务只能再对A加S锁，而不能加X锁，即其他事务只能读A，而不能修改；
- 当T2为修改A而申请对A的X锁时被拒绝只能等待T1释放A上的锁
- T1再读A，这时读出的A仍是100，即可重复读
- T1结束才释放A上的S锁。T2才获得对A的X锁

封锁协议小结

- 三级协议的主要区别
  - 什么操作需要申请封锁以及何时释放锁（即持锁时间）
- 不同的封锁协议使事务达到的一致性级别不同
  - 封锁协议级别越高，一致性程度越高

	X锁		S锁		消除数据异常		
	操作结束释放	事务结束释放	操作结束释放	事务结束释放	不脏写	不读“脏”数据	可重复读
一级封锁协议		√			√		
二级封锁协议		√	√		√	√	
三级封锁协议		√		√	√	√	√

表11.1 不同级别的封锁协议和一致性保证

第11章 并发控制

- 11.1 两阶段封锁协议
- 11.2 时间戳排序协议
- 11.3 乐观并发控制协议
- 11.4 三级封锁协议
- 11.5 封锁粒度

封锁粒度

- 封锁对象的大小称为封锁粒度(Granularity)
- 封锁的对象:逻辑单元，物理单元

  例：在关系数据库中，封锁对象：

  - 逻辑单元: 属性值、属性值的集合、元组、关系、索引项、整个索引、整个数据库等
  - 物理单元：页（数据页或索引页）、物理记录等

选择封锁粒度原则

- 封锁粒度与系统的并发度和并发控制的开销密切相关。
  - 封锁的粒度越大，数据库所能够封锁的数据单元就越少，并发度就越小，系统开销也越小；
  - 封锁的粒度越小，并发度较高，但系统开销也就越大

例：事务T1需要修改元组L1，事务T2需要修改元组L2，L1和L2位于同一个数据页面A。

- 若封锁粒度是数据页，事务T1需要修改元组L1，则T1必须对包含L1的整个数据页A加锁。如果T1对A加锁后事务T2要修改A中元组L2，则T2被迫等待，直到T1释放A。
- 如果封锁粒度是元组，则T1和T2可以同时对L1和L2加锁，不需要互相等待，提高了系统的并行度。

封锁粒度越小，并发度越高

又如，事务T3需要读取整个表，若封锁粒度是元组，T3必须对表中的每一个元组加锁，大表的情况下开销极大；若封锁粒度是关系，T3只需要一次加锁，开销降低；若封锁粒度是数据页，事务T3需要多次加锁，因为一个数据页通常包含多个元组，开销介于按元组和按关系封锁的开销之间。

封锁粒度越小，封锁开销就越大。

选择封锁粒度的原则（续）

- 多粒度封锁(Multiple Granularity Locking)

  在一个系统中同时支持多种封锁粒度供不同的事务选择

- 选择封锁粒度

  同时考虑封锁开销和并发度两个因素, 适当选择封锁粒度

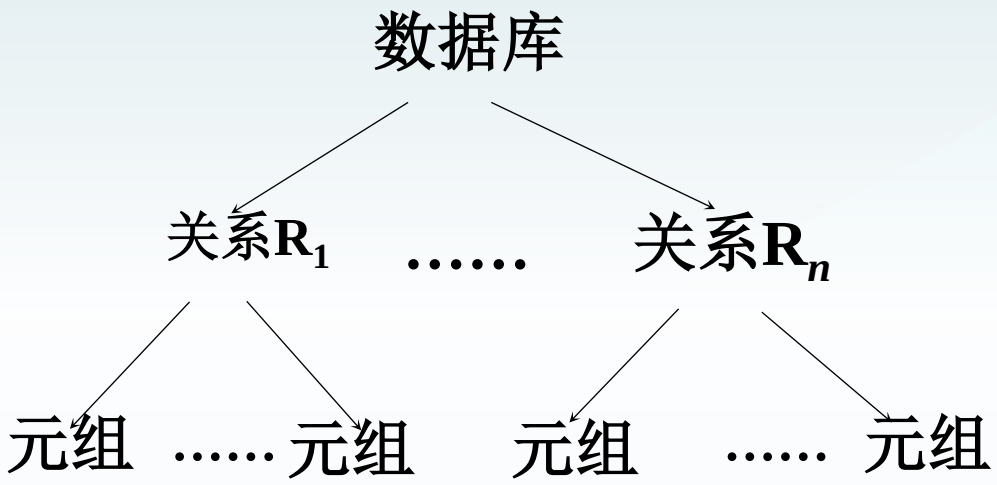
  - 需要处理多个关系的大量元组的用户事务：以数据库为封锁单位
  - 需要处理大量元组的用户事务：以关系为封锁单元
  - 只处理少量元组的用户事务：以元组为封锁单位

多粒度封锁

- 多粒度树
  - 以树形结构来表示多级封锁粒度
  - 根结点是整个数据库，表示最大的数据粒度
  - 叶结点表示最小的数据粒度

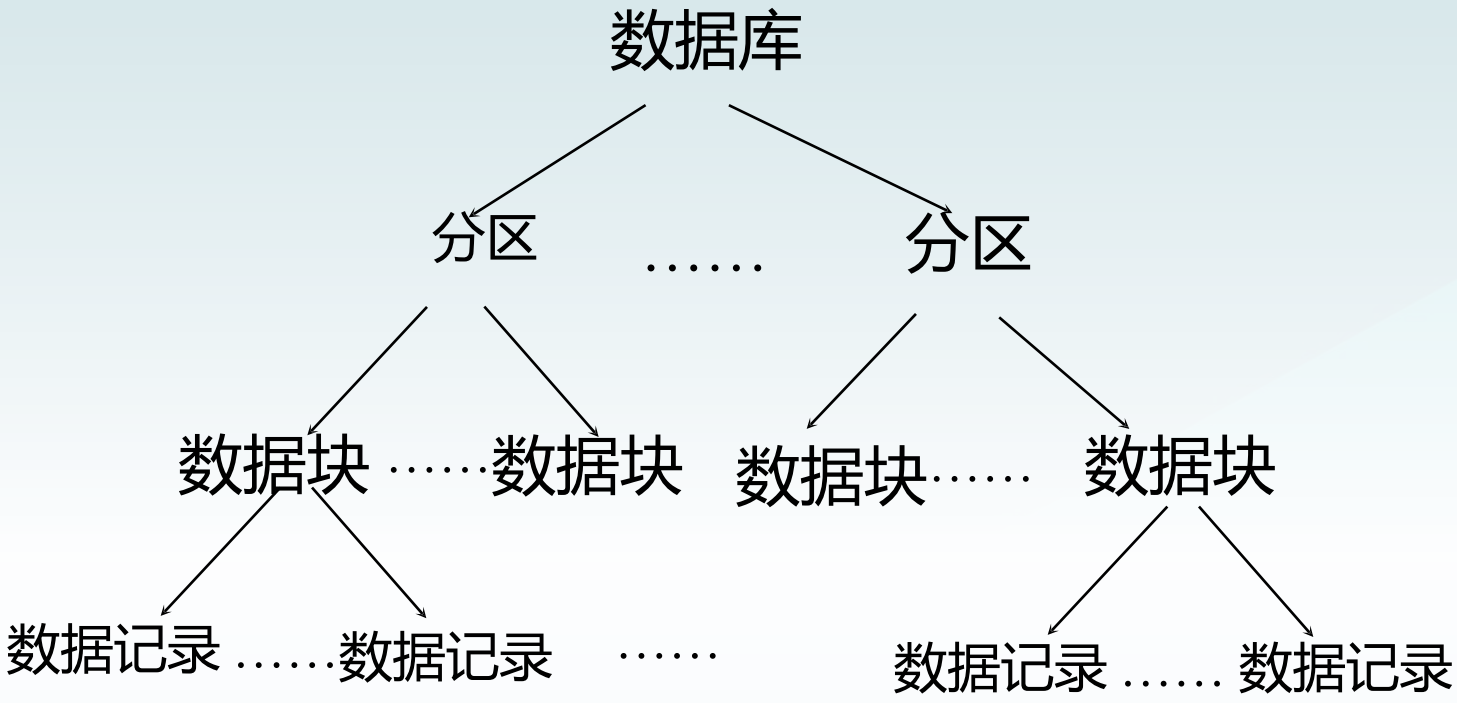
多粒度封锁（续）

例：三级粒度树。根结点为数据库，数据库的子结点为关系，关系的子结点为元组。



三级粒度树

例2：四级粒度树。



四级粒度树

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
- 在多粒度封锁中一个数据对象可能以两种方式封锁：显式封锁和隐式封锁

显式封锁和隐式封锁

- 显式封锁: 直接加到数据对象上的封锁
- 隐式封锁:是该数据对象没有独立加锁，是由于其上级结点加锁而使该数据对象加上了锁
- 显式封锁和隐式封锁的效果是一样的

显式封锁和隐式封锁（续）

- 系统检查封锁冲突时
  - 要检查显式封锁
  - 还要检查隐式封锁
- 例如事务T要对关系R_1加X锁
  - 系统必须搜索其上级结点数据库、关系R_1
  - 还要搜索R_1的下级结点，即R_1中的每一个元组
  - 如果其中某一个数据对象已经加了不相容锁，则T必须等待

显式封锁和隐式封锁（续）

- 对某个数据对象加锁，系统要检查
  - 该数据对象
    - 有无显式封锁与之冲突
  - 所有上级结点
    - 检查本事务的显式封锁是否与该数据对象上的隐式封锁冲突：（由上级结点已加的封锁造成的）
  - 所有下级结点
    - 看上面的显式封锁是否与本事务的隐式封锁（将加到下级结点的封锁）冲突

12.8 封锁的粒度

12.8.1 多粒度封锁

12.8.2 意向锁

12.8.2 意向锁

- 引进意向锁（intention lock）目的
  - 有了意向锁，DBMS无须逐个检查下一级结点的显式封锁
  - 提高对某个数据对象加锁时系统的检查效率

意向锁(续)

- 如果对一个结点加意向锁，则说明该结点的下层结点正在被加锁
- 对任一结点加基本锁，必须先对它的上层结点加意向锁
- 例如，对任一元组加锁时，必须先对它所在的数据库和关系加意向锁

常用意向锁

- 意向共享锁(Intent Share Lock，简称IS锁)
- 意向排它锁(Intent Exclusive Lock，简称IX锁)
- 共享意向排它锁(Share Intent Exclusive Lock，简称SIX锁)

意向锁（续）

- IS锁
  - 如果对一个数据对象加IS锁，表示它的后裔结点拟（意向）加S锁。

例如：事务T_1要对R_1中某个元组加S锁，则要首先对关系R_1和数据库加IS锁

意向锁（续）

- IX锁
  - 如果对一个数据对象加IX锁，表示它的后裔结点拟（意向）加X锁。

例如：事务T_1要对R_1中某个元组加X锁，则要首先对关系R_1和数据库加IX锁

意向锁（续）

- SIX锁
  - 如果对一个数据对象加SIX锁，表示对它加S锁，再加IX锁，即SIX = S + IX。

例：对某个表加SIX锁，则表示该事务要读整个表（所以要对该表加S锁），同时会更新个别元组（所以要对该表加IX锁）。

意向锁（续）

意向锁的相容矩阵

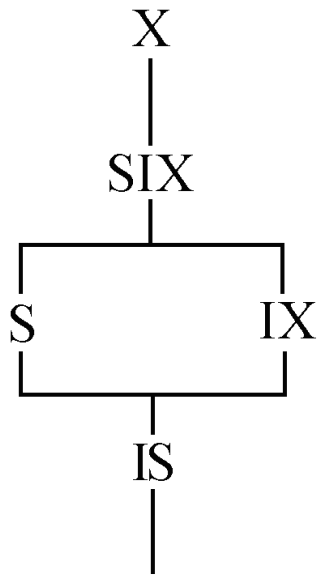
T_1 \ T_2	S	X	IS	IX	SIX	–
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
–	Y	Y	Y	Y	Y	Y

Y=Yes，表示相容的请求　　N=No，表示不相容的请求

(a) 数据锁的相容矩阵

意向锁（续）

- 锁的强度
  - 锁的强度是指它对其他锁的排斥程度
  - 一个事务在申请封锁时以强锁代替弱锁是安全的，反之则不然



(b) 锁的强度的偏序关系

意向锁（续）

- 具有意向锁的多粒度封锁方法
  - 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

例如：事务T_1要对关系R_1加S锁

- 要首先对数据库加IS锁
- 检查数据库和R_1是否已加了不相容的锁(X或IX)
- 不再需要搜索和检查R_1中的元组是否加了不相容的锁(X锁)

意向锁（续）

- 具有意向锁的多粒度封锁方法
  - 提高了系统的并发度
  - 减少了加锁和解锁的开销
  - 在实际的数据库管理系统产品中得到广泛应用

课堂测试

1.未引入意向锁之前，对某个数据对象加锁，系统需要检查（）

A. 该数据对象　B. 该数据对象所有上级节点

C.整个数据库　　D.该数据对象所有下级节点

总结

- 事务具有ACID特性，即原子性、一致性、隔离性和持续性。并发控制技术用来保证并发事务的隔离性（I）和一致性（D）。
- 以“什么时候定序”、“如何定序”、“什么时候检验”、“检验内容是什么”、“检验不通过如何处理”这些问题为主线，本章分别介绍了两阶段封锁协议、时间戳排序协议、乐观并发控制协议。
- 并发事务调度中可能存在的级联回滚、死锁和活锁的处理方法。
- 在并发控制实现技术中，并发事务要求的隔离级别越高，系统的性能往往越低。系统默认的隔离级别并不是可串行化的，有些甚至连可串行化隔离级别都不支持。需要掌握三级封锁协议的实现技术以及其是如何确保系统能够达到指定的隔离级别，了解多版本并发控制的基本实现技术及其可能存在的写偏序数据异常。