数据库系统概论

An Introduction to Database System

第十一章　并发控制

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

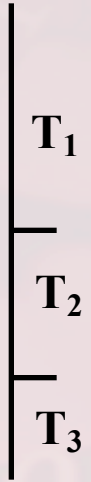
- 飞机订票数据库系统
- 银行数据库系统

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（续）

❖多事务执行方式

（1）事务串行执行

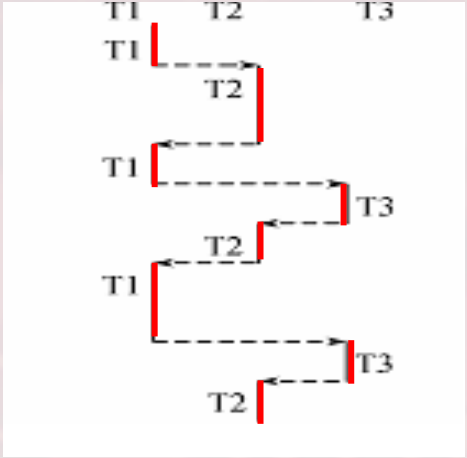


- 每个时刻只有一个事务运行，其他事务必须等到这个事务结束以后方能运行
- 不能充分利用系统资源，发挥数据库共享资源的特点

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



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

❖ 本章讨论的数据库系统并发控制技术是以单处理机系统为基础的

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

T_1	T_2
① 读A=16	
②	读A=16
③ A←A-1	
写回A=15	
④	A←A-3
	写回A=13

T1的修改被T2覆盖了！

第十一章 并发控制

11.1 并发控制概述
11.2 封锁
11.3 封锁协议
11.4 活锁和死锁
11.5 并发调度的可串行性
11.6 两段锁协议
11.7 封锁的粒度
*11.8 其他并发控制机制
11.9 小结

11.1 并发控制概述

- 事务是并发控制的基本单位
- 并发控制机制的任务
  - 对并发操作进行正确调度
  - 保证事务的隔离性
  - 保证数据库的一致性

并发控制（续）

并发操作带来数据的不一致性实例

[例11.1]飞机订票系统中的一个活动序列

① 甲售票点(事务T_1)读出某航班的机票余额A，设A=16；

② 乙售票点(事务T_2)读出同一航班的机票余额A，也为16；

③ 甲售票点卖出一张机票，修改余额A←A-1，所以A为15，把A写回数据库；

④ 乙售票点卖出三张机票，修改余额A←A-3，所以A为13，把A写回数据库

- 结果明明卖出4张机票，数据库中机票余额只减少3

T_1	T_2
① 读A=16	
②	读A=16
③ A←A-1	
写回A=15	
④	A←A-3
	写回A=13

T1的修改被T2覆盖了！

并发控制概述（续）

- 并发操作带来的数据不一致性
  1. 丢失修改（Lost Update）
  2. 不可重复读（Non-repeatable Read）
  3. 读“脏”数据（Dirty Read）

- 记号
  - R(x): 读数据x
  - W(x): 写数据x

1. 丢失修改

❖两个事务T_1和T_2读入同一数据并修改，T_2的提交结果破坏了T_1提交的结果，导致T_1的修改被丢失。

	T_1	T_2
①	R(A)=16	
②		R(A)=16
③	A←A-1	
	W(A)=15	
④		A←A-3
		W(A)=13

2. 不可重复读

❖不可重复读是指事务T_1读取数据后，事务T_2执行更新操作，使T_1无法再现前一次读取结果。

不可重复读（续）

❖不可重复读包括三种情况：

（1）情况1

- 事务1读取某一数据
- 事务2对其做了修改
- 当事务1再次读该数据时，得到与前一次不同的值

不可重复读（续）

例如：

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)	

读一修改

- T_1读取B=100进行运算
- T_2读取同一数据B，对其进行修改后将B=200写回数据库。
- T_1为了对读取值校对重读B，B已为200，与第一次读取值不一致

不可重复读（续）

（2）情况2

- 事务T_1按一定条件从数据库中读取了某些数据记录
- 事务T_2删除了其中部分记录
- 当T_1再次按相同条件读取数据时，发现某些记录神秘地消失了。

不可重复读（续）

（3）情况3

- 事务T_1按一定条件从数据库中读取某些数据记录
- 事务T_2插入了一些记录
- 当T_1再次按相同条件读取数据时，发现多了一些记录

后两种不可重复读有时也称为幻影现象（Phantom Row）

3. 读“脏”数据

读“脏”数据是指：

- 事务T_1修改某一数据，并将其写回磁盘
- 事务T_2读取同一数据后，T_1由于某种原因被撤销
- 这时T_1已修改过的数据恢复原值，T_2读到的数据就与数据库中的数据不一致
- T_2读到的数据就为“脏”数据，即不正确的数据

读“脏”数据（续）

例如：

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	

修改－读

- T_1将C值修改为200
- T_2读到C为200
- T_1由于某种原因撤销，其修改作废，C恢复原值100 。
- 这时T_2读到的C为200，与数据库内容不一致，就是“脏”数据

并发控制概述（续）

- 数据不一致性：由于并发操作破坏了事务的隔离性
- 并发控制就是要用正确的方式调度并发操作，使一个用户事务的执行不受其他事务的干扰，从而避免造成数据的不一致性
- 对数据库的应用有时允许某些不一致性，可以降低对一致性的要求以减少系统开销

并发控制概述（续）

- 并发控制的主要技术
  - 封锁(Locking)
  - 时间戳(Timestamp)
  - 乐观控制法
  - 多版本并发控制(MVCC)

小结

- 并发操作带来的数据不一致性
  1. 丢失修改 （修改-修改冲突）
  2. 不可重复读 （读-更新冲突）
     - 修改
     - 删除
     - 插入
  3. 读“脏”数据（修改-读冲突）

第十一章 并发控制

11.1 并发控制概述
11.2 封锁
11.3 封锁协议
11.4 活锁和死锁
11.5 并发调度的可串行性
11.6 两段锁协议
11.7 封锁的粒度
*11.8 其他并发控制机制
11.9 小结

11.2 封锁

- 什么是封锁
- 基本封锁类型
- 锁的相容矩阵

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。

T_1	T_2
① R(A)=16	
②	R(A)=16
③ A←A-1	
W(A)=15	
④	A←A-3
	W(A)=13

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

- 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
- 排它锁又称为写锁

共享锁

- 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
- 共享锁又称为读锁

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_2	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

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_2 / T_1	X	S	–
X	N	N	Y
S	N	Y	Y
–	Y	Y	Y

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_1 \ T_2	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

锁的相容矩阵

T_1 \ T_2	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

第十一章 并发控制

11.1 并发控制概述
11.2 封锁
11.3 封锁协议
11.4 活锁和死锁
11.5 并发调度的可串行性
11.6 两段锁协议
11.7 封锁的粒度
*11.8 其他并发控制机制
11.9 小结

11.3 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，在不同的程度上保证并发操作的正确调度。

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。

使用封锁机制解决丢失修改问题

例：

	T_1	T_2
①	R(A)=16	
②		R(A)=16
③	A←A-1	
	W(A)=15	
④		A←A-3
		W(A)=13

	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-3
⑤		W(A)=12
		Commit
	没有丢失修改	Unlock A

一级封锁协议（续）

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

不可重复读

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

T_1	T_2
① Xlock C	
获得	
② R(C)=100	
C←C*2	
W(C)=200	
③	R(C)=200
④ Rollback	
C恢复为100	
Unlock C	

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。

使用二级封锁协议解决丢失修改问题

例：

	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-3
⑤		W(A)=12
		Commit
		Unlock A

没有丢失修改

- 事务T1在读A进行修改之前先对A加X锁
- 当T2再请求对A加X锁时被拒绝
- T2只能等待T1释放A上的锁后T2获得对A的X锁
- 这时T2读到的A已经是T1更新过的值15
- T2按此新的A值进行运算，并将结果值A=14送回到磁盘。避免了丢失T1的更新。

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	R(C)=200
③ ROLLBACK	
C恢复为100	

读“脏”数据

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
	Commit C
	Unlock C

未读“脏”数据

二级封锁协议（续）

- 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: 不可重复读

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 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。

使用封锁机制解决不可重复读问题

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

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

4. 封锁协议小结

	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		√		√	√	√	√

不同级别的封锁协议和一致性保证

- 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

小结

- 基本封锁类型
  - X锁
  - S锁
- 封锁协议
  - 一级封锁协议
  - 二级封锁协议
  - 三级封锁协议

第十一章 并发控制

11.1 并发控制概述
11.2 封锁
11.3 封锁协议
11.4 活锁和死锁
11.5 并发调度的可串行性
11.6 两段锁协议
11.7 封锁的粒度
*11.8 其他并发控制机制
11.9 小结

11.4 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

11.4 活锁和死锁

11.4.1 活锁

11.4.2 死锁

11.4.1 活锁

T_1	T_2	T_3	T_4
Lock R	•	•	•
	•	•	•
	•	•	•
•	Lock R		
•	等待	Lock R	
•	等待	等待	
Unlock R	等待	•	
	等待	Lock R	
•	等待	•	Lock R
•	等待	•	等待
•	等待	Unlock R	•
	等待	•	Lock R
	等待		•

11.4.1 活锁

T_1	T_2	T_3	T_4
Lock R	•	•	•
	•	•	•
	•	•	•
•	Lock R		
•	等待	Lock R	
•	等待	等待	
Unlock R	等待	•	
	等待	Lock R	
•	等待	•	Lock R
•	等待	•	等待
•	等待	Unlock R	•
	等待	•	Lock R
	等待		•

11.4.1 活锁

T_1	T_2	T_3	T_4
Lock R	•	•	•
	•	•	•
	•	•	•
•	Lock R		
•	等待	Lock R	
•	等待	等待	
Unlock R	等待	•	
	等待	Lock R	
•	等待	•	Lock R
•	等待	•	等待
•	等待	Unlock R	•
	等待	•	Lock R
	等待		•

活锁

活锁（续）

- 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

11.4 活锁和死锁

11.4.1 活锁

11.4.2 死锁

死锁（续）

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
•	•
•	•
•	•

死锁（续）

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
•	•
•	•
•	•

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	Lock R_1
等待	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 R2
	Lock R_1
	•
	•

（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	Lock R_1
等待	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 R2
	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

    数据库系统中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。
  - 难以实现

    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

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

❖死锁的诊断

（1）超时法

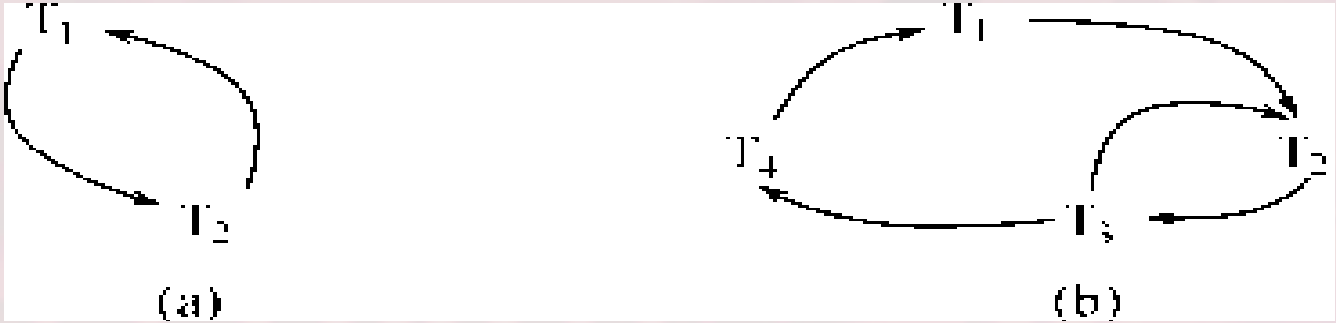
（2）等待图法

（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

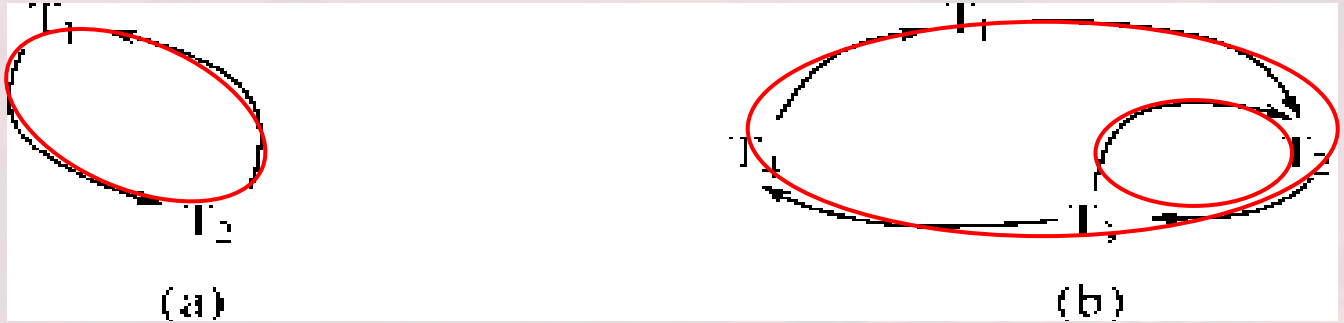
（2）等待图法

- 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



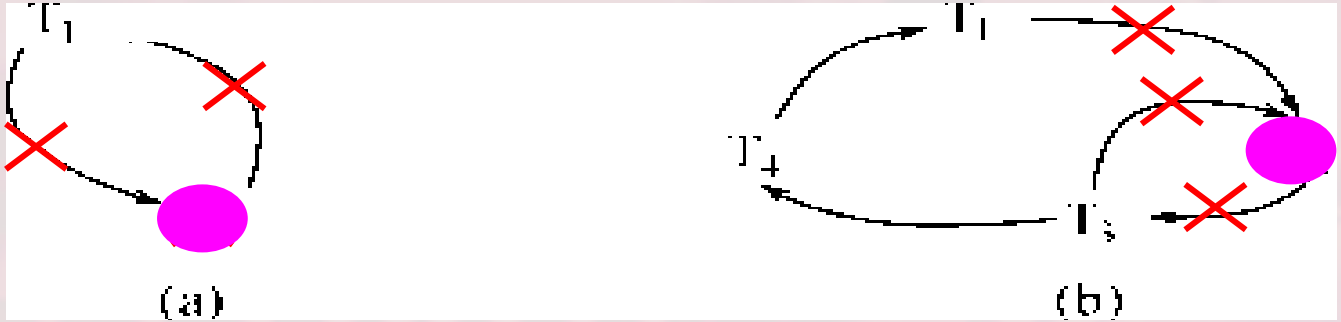
等待图法（续）

- 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。



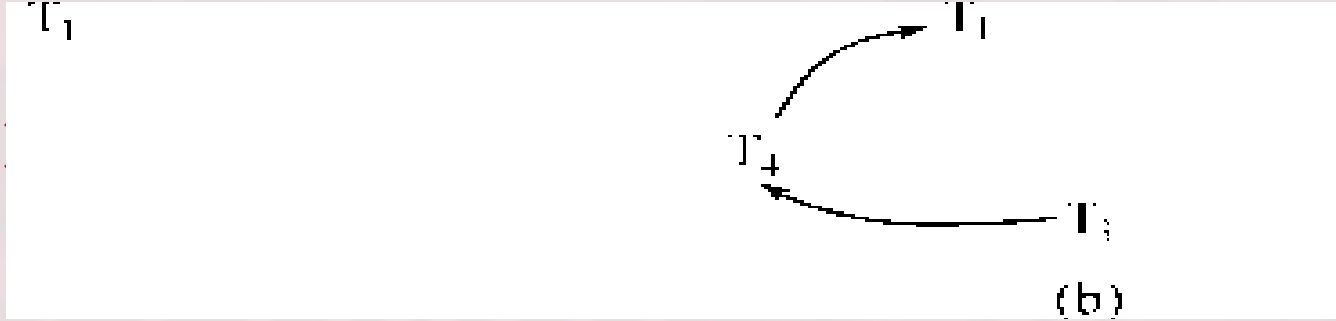
死锁的诊断与解除（续）

- 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



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



小结

- 活锁
  - 解决方法：先来先服务
- 死锁
  - 预防
    - 一次封锁法
    - 顺序封锁法
  - 检测与解除
    - 超时法
    - 等待图法

第十一章 并发控制

11.1 并发控制概述
11.2 封锁
11.3 封锁协议
11.4 活锁和死锁
11.5 并发调度的可串行性
11.6 两段锁协议
11.7 封锁的粒度
*11.8 其他并发控制机制
11.9 小结

11.5 并发调度的可串行性

- 数据库管理系统对并发事务不同的调度可能会产生不同的结果
- 串行调度是正确的
- 执行结果等价于串行调度的调度也是正确的，称为可串行化调度

11.5 并发调度的可串行性

11.5.1 可串行化调度

11.5.2 冲突可串行化调度

11.5.1 可串行化调度

- 可串行化(Serializable)调度
  - 多个事务的并发执行是正确的，当且仅当其结果与按某一次序串行地执行这些事务时的结果相同
- 可串行性(Serializability)
  - 是并发事务正确调度的准则
  - 一个给定的并发调度，当且仅当它是可串行化的，才认为是正确调度

可串行化调度（续）

[例] 现在有两个事务，分别包含下列操作：

- 事务T1：读B；A=B+1；写回A
- 事务T2：读A；B=A+1；写回B

现给出对这两个事务不同的调度策略

串行调度,正确的调度

T_1	T_2
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
A=Y+1=3	
W(A)	
Unlock A	
	Slock A
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

- 假设A、B的初值均为2。
- 按$T_1 \rightarrow T_2$次序执行结果为A=3，B=4
- 串行调度策略,正确的调度

串行调度,正确的调度

T_1	T_2
	Slock A
	X=R(A)=2
	Unlock A
	Xlock B
	B=X+1=3
	W(B)
	Unlock B
Slock B	
Y=R(B)=3	
Unlock B	
Xlock A	
A=Y+1=4	
W(A)	
Unlock A	

- 假设A、B的初值均为2。
- $T_2 \rightarrow T_1$次序执行结果为

  B=3， A=4
- 串行调度策略,正确的调度

不可串行化调度，错误的调度

T_1	T_2
Slock B	
Y=R(B)=2	
	Slock A
	X=R(A)=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1=3	
W(A)	
	Xlock B
	B=X+1=3
	W(B)
Unlock A	
	Unlock B

- 执行结果A=3，B=3，

  与(a)、(b)的结果都不同

- 是错误的调度

可串行化调度，正确的调度

T_1	T_2
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1=3	等待
W(A)	等待
Unlock A	等待
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

- 执行结果A=3, B=4，与第一种串行调度的执行结果相同
- 是正确的调度

四种调度

T_1	T_2
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
A=Y+1=3	
W(A)	
Unlock A	
	Slock A
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

串行调度

T_1	T_2
	Slock A
	X=R(A)=2
	Unlock A
	Xlock B
	B=X+1=3
	W(B)
	Unlock B
Slock B	
Y=R(B)=3	
Unlock B	
Xlock A	
A=Y+1=4	
W(A)	
Unlock A	

串行调度

T_1	T_2
Slock B	
Y=R(B)=2	
	Slock A
	X=R(A)=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1=3	
W(A)	
	Xlock B
	B=X+1=3
	W(B)
Unlock A	
	Unlock B

不可串行化的调度

T_1	T_2
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1=3	等待
W(A)	等待
Unlock A	等待
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

可串行化的调度

11.5 并发调度的可串行性

11.5.1 可串行化调度

11.5.2 冲突可串行化调度

11.5.2 冲突可串行化调度

- 冲突可串行化
  - 一个比可串行化更严格的条件

- 冲突操作：是指不同的事务对同一数据的读写操作和写写操作：

  $R_i(x)$与$W_j(x)$　　/*事务T_i读x，T_j写x，其中i≠j*/

  $W_i(x)$与$W_j(x)$　　/*事务T_i写x，T_j写x，其中i≠j*/

涉及同一个数据库元素，并且至少有一个是写操作

冲突

- 不冲突操作
  - $r_i(X)$; $r_j(X)$ 读
  - $r_i(X)$; $w_j(Y)$, X不等于Y
  - $w_i(X)$; $r_j(Y)$, X不等于Y
  - $w_i(X)$; $w_j(Y)$, X不等于Y

冲突

- 不能交换（Swap）的动作:
  - 同一事务的两个操作
  - 不同事务的冲突操作
    - $R_i(x)$与$W_j(x)$
    - $W_i(x)$与$W_j(x)$

冲突可串行化

- 一个调度S_c在保证冲突操作的次序不变的情况下，通过交换两个事务不冲突操作的次序得到另一个调度S_c'，如果S_c'是串行的，称调度S_c是冲突可串行化的调度
- 若一个调度是冲突可串行化，则一定是可串行化的调度

冲突可串行化（续）

[例] 今有3个事务的一个调度

r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

判断该调度是否是冲突可串行化的调度。

Sc1 = r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

冲突可串行化（续）

[例] 今有3个事务的一个调度

r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

判断该调度是否是冲突可串行化的调度。

Sc1 = r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

r3(B) w3(B) r1(A) r2(B) r2(A) w2(B) r1(B) w1(A)

冲突可串行化（续）

[例] 今有3个事务的一个调度

r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

判断该调度是否是冲突可串行化的调度。

Sc1 = r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

r3(B) w3(B) r1(A) r2(B) r2(A) w2(B) r1(B) w1(A)

冲突可串行化（续）

[例] 今有3个事务的一个调度

r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

判断该调度是否是冲突可串行化的调度。

Sc1 = r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

r3(B) w3(B) r1(A) r2(B) r2(A) w2(B) r1(B) w1(A)

r3(B) w3(B) r2(B) r2(A) w2(B) r1(A) r1(B) w1(A)

冲突可串行化（续）

[例] 今有3个事务的一个调度

r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

判断该调度是否是冲突可串行化的调度。

Sc1 = r3(B) r1(A) w3(B) r2(B) r2(A) w2(B) r1(B) w1(A)

r3(B) w3(B) r1(A) r2(B) r2(A) w2(B) r1(B) w1(A)
r3(B) w3(B) r2(B) r2(A) w2(B) r1(A) r1(B) w1(A)

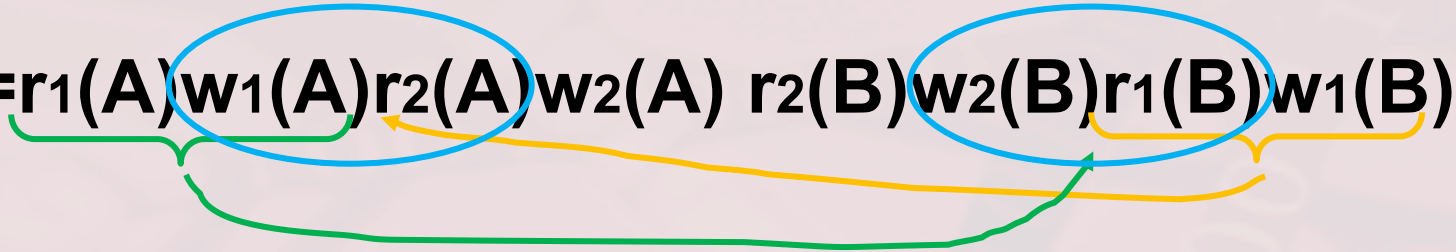
Sc2 = r3(B) w3(B) | r2(B) r2(A) w2(B) | r1(A) r1(B) w1(A)

所以 **Sc1**是冲突可串行化的调度。

冲突可串行化（续）

例：

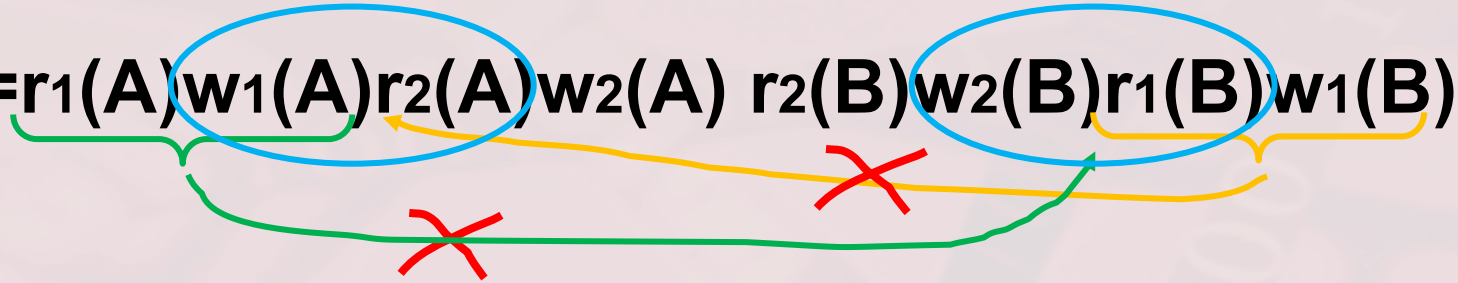
$Sd=r_1(A)w_1(A)r_2(A)w_2(A)\ r_2(B)w_2(B)r_1(B)w_1(B)$



冲突可串行化（续）

例：

$Sd=r_1(A)w_1(A)r_2(A)w_2(A)\ r_2(B)w_2(B)r_1(B)w_1(B)$



- 不能通过无冲突交换将Sd变换为串行调度
- Sd不是冲突可串行化的调度

冲突可串行化调度

- 冲突可串行化调度是可串行化调度的充分条件，不是必要条件。还有不满足冲突可串行化条件的可串行化调度。

[例11.4]有3个事务

$T_1=W_1(Y)W_1(X)$，$T_2=W_2(Y)W_2(X)$，$T_3=W_3(X)$

- 调度$L_1=W_1(Y)W_1(X)W_2(Y)W_2(X)\ W_3(X)$是一个串行调度。
- 调度$L_2=W_1(Y)W_2(Y)W_2(X)W_1(X)W_3(X)$不满足冲突可串行化。

  但是调度L_2是可串行化的，因为L_2执行的结果与调度L_1相同，Y的值都等于T_2的值，X的值都等于T_3的值

小结

- 可串行化调度
- 冲突可串行化调度
- 可串行化调度与冲突可串行化调度之间的关系

第十一章 并发控制

11.1 并发控制概述
11.2 封锁
11.3 封锁协议
11.4 活锁和死锁
11.5 并发调度的可串行性
11.6 两段锁协议
11.7 封锁的粒度
*11.8 其他并发控制机制
11.9 小结

11.6 两段锁协议

- 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

两段锁协议（续）

- “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

两段锁协议（续）

例

事务T_i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　　→|

事务T_j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；

两段锁协议（续）

例

事务T_i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；

|← 收缩阶段 →|

事务T_j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；

两段锁协议（续）

例

事务T_i遵守两段锁协议，其封锁序列是：

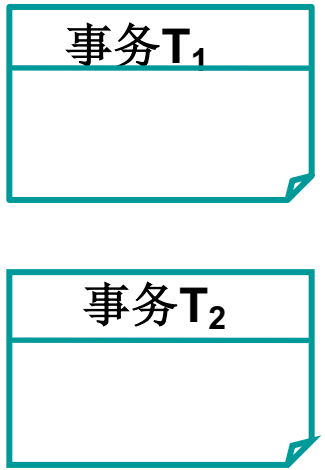
Slock A　Slock B　Xlock C　Unlock B　Unlock A　Unlock C；

事务T_j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；

两段锁协议（续）

	事务T_1	事务T_2
扩展阶段	Slock(A)	
	R(A)=260	
		Xlock(B)
		W(B)=300
	Xlock(C)	
	W(C)=160	
		Xlock(A)
		等待
	Unlock(A)	等待
		W(A)=250
收缩阶段		Unlock(B)
	Unlock(C)	
		Unlock(A)



- 遵守两段锁协议，是一个可串行化调度。
- 如何验证？

两段锁协议（续）

事务T_1	事务T_2
Slock(A)	
R(A)=260	
	Xlock(B)
	W(B)=300
Xlock(C)	
W(C)=160	
	Xlock(A)
	等待
Unlock(A)	等待
	W(A)=250
	Unlock(B)
Unlock(C)	
	Unlock(A)

$L_1=R_1(A)W_2(B)W_1(C)W_2(A)$

$L_S=R_1(A)\ W_1(C)\ W_2(B)W_2(A)$

两段锁协议（续）

- 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，不一定所有事务都符合两段锁协议

两段锁协议（续）

T_1	T_2
Slock B	
Y=R(B)=2	
Unlock B	
Xlock A	
	Slock A
A=Y+1=3	等待
W(A)	等待
Unlock A	等待
	X=R(A)=3
	Unlock A
	Xlock B
	B=X+1=4
	W(B)
	Unlock B

- 对**T1**和**T2**的调度没有遵守两段锁协议
- 但是它是可串行化的

$$L_1=R_1(B)W_1(A)R_2(A)W_2(B)$$

两段锁协议（续）

T_1	T_2
Slock B	
Y=R(B)=2	
	Slock A
	X=R(A)=2
Unlock B	
	Unlock A
Xlock A	
A=Y+1=3	
W(A)	
	Xlock B
	B=X+1=3
	W(B)
Unlock A	
	Unlock B

- 对T1和T2的调度没有遵守两段锁协议
- 它不是可串行化的

- 并发事务遵守两段锁协议，对这些事务的任何并发调度策略都是可串行化的；
- 不遵守两段锁协议，对这些事务的并发调度策略可能是可串行化的，也可能不是可串行化的。

两段锁协议（续）

- 两段锁协议与防止死锁的一次封锁法
  - 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，因此一次封锁法遵守两段锁协议
  - 但是两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议的事务可能发生死锁

两段锁协议（续）

[例] 遵守两段锁协议的事务可能发生死锁

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

死锁

小结

- 两段锁协议
  - 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