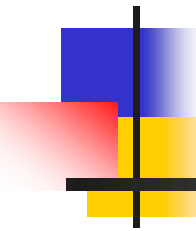


第9章 并发控制

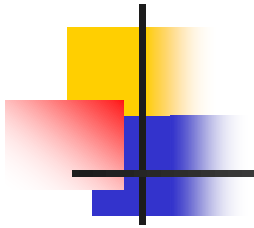
北京理工大学 计算机学院
张文耀
zhwenyao@bit.edu.cn

主要内容

- 并发控制概述
- **9.1** 并发事务运行存在的异常问题
- **9.2** 并发调度的可串行性
- **9.3** 基于封锁的并发控制技术
- **9.4** 多粒度封锁
- **9.5** 基于时间戳协议的并发控制
- **9.6** 基于有效性确认的并发控制
- **9.7** 插入与删除操作对并发控制的影响
- **9.8 SQL Server**中的并发控制
- 小结

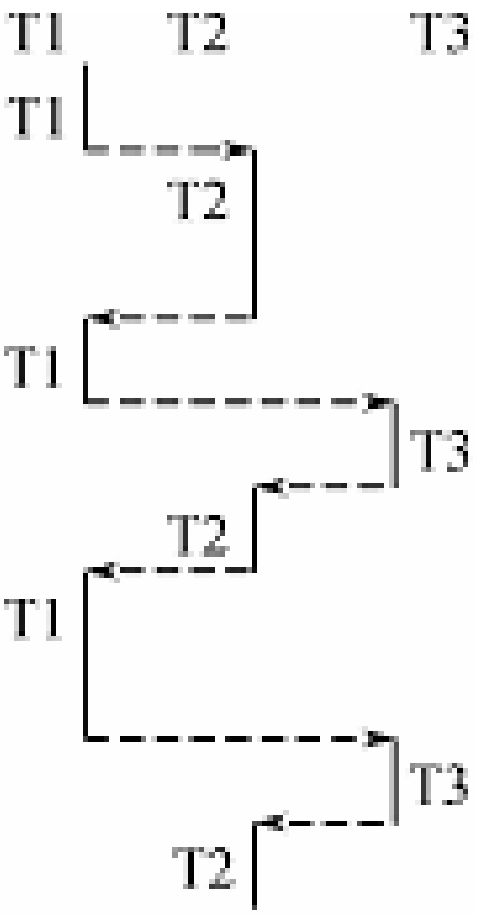
并发控制概述

- 多用户数据库系统
  - 数据资源共享
  - 外在表现：多个用户同时使用的数据库系统
  - 系统内部：允许多个事务以并发的方式运行
- 多事务执行方式
  - 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和数据库资源。



- 交叉并发方式（**interleaved concurrency**）
  - 适应于单处理机系统；
  - 并行事务的并行操作轮流交叉运行；
  - 能够减少处理机的空闲时间，提高系统的效率。
- 同时并发方式（**simultaneous concurrency**）
  - 适应于多处理机系统，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；
  - 理想的并发方式，但受制于硬件环境，以及更为复杂的实现机制的限制。

- 本章所讨论的是：
  - 单处理机系统上多个事务的并发执行
  - 串行执行（X）
  - 交叉执行（V）



- 事务并发执行带来的问题

  可能会存取和存储不正确的数据，破坏事务的隔离性和数据库的一致性

- **DBMS必须**

  提供一种控制机制，以解决并发运行事务带来的异常问题

- 并发控制

  保证并发执行的事务使数据库状态保持一致性的过程

- 并发控制技术
  - 基于封锁的技术，基于时间戳的技术，...
  - 衡量一个数据库管理系统性能的重要标志之一

9.1 并发执行的异常问题

- 并发操作带来的数据不一致性

  1）丢失更新（**lost update**）

  2）不可重复读**(non-repeatable read)**

  3）读“脏”数据**(dirty read)**

1）丢失更新

时间	更新事务 T_1	机票余额 R	更新事务 T_2
t0		100	
t1	Read (R)		
t2			Read(R)
t3	R:=R-1=99		
t4			R:=R-1=99
t5	Update(R)		
t6		99	Update(R)
t7		99	

图 9-1 机票订购事务中的丢失更新问题

- 丢失更新
  - 事务1与事务2从数据库中读入同一数据并修改；
  - 事务2的提交结果破坏了事务1提交的结果，导致事务1的修改被丢失。
- 丢失更新——怎么办？

2）不可重复读

时间	事务 T_1	帐户 A 余额（万）	事务 T_2
t0		10	
t1	Read (A=10)		
t2			Read (A =10)
t3			A：=A-1
t4			Update(A)
t5	Read (A)	9	

图 9-2 转账事务不能重复读数据问题

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

- 不可重复读
  - 事务**T1**读取数据后，**T2**对同一数据执行更新操作，使**T1**再次读取该数据时，得到与前一次不同的值。
- 三类不可重复读的情况
  - 事务**1**读取某一数据后， **T2**对其做了修改，当**T1**再次读该数据时，得到与前一次不同的值；
  - **T2**删除了部分记录，当**T1**再次读取数据时，某些记录消失；
  - **T2**插入了一些记录，当**T1**再次按相同条件读数据时，多了一些记录；
  - 后两种有时也称为幻影（**phantom**）现象。

3）读“脏”数据

时间	事务 T_1	订单数 A	事务 T_2
t0		200	
t1	Read (A)		
t2	A=300		
t3	Update (A)		
t4			Read (A)
t5			A=300
t6	ROLLBACK		
t7		200	

图 9-3 商品订购事务中读脏数据问题

T_1	T_2
① 读C=100 C←C*2 写回C	
②	读C=200
③ ROLLBACK C恢复为100	

- 读脏数据
  - 事务T1修改某一数据，并将其写回磁盘；
  - 事务T2读取同一数据后，事务T1由于某种原因被撤消，这时被事务T1修改过的数据恢复原值；
  - 事务T2读到的数据就与数据库中的数据不一致，是不正确的数据，又称为“脏”数据。

- 并发操作带来的数据不一致性
  - 丢失更新（**lost update**）
  - 不可重复读**(non-repeatable read)**
  - 读“脏”数据**(dirty read)**
- 这种数据库的不一致性是由并发操作引起的，主要原因是并发操作破坏了事务的隔离性。
- 并发控制机制的任务：
  - 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，从而避免数据的不一致性
    - 保证事务的隔离性
    - 保证数据库的一致性

9.2 并发调度的可串行性

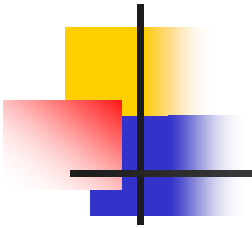
- 调度
  - 并发事务的操作顺序称之为调度（**schedule**）；
  - 调度是针对多个并发执行的事务而言的；
  - 一个调度是一个操作序列（包括了所涉及的并发事务的所有操作）；
  - 调度中某个事务的操作顺序必须保持与该事务原有的顺序相同；
  - 并发事务的调度不是唯一的；
  - 不同的事务调度可能会产生不同的结果。

事务调度示例

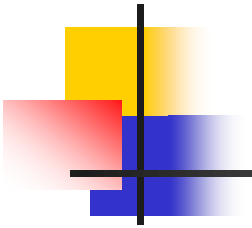
T1	T2	A	B
Read(A)		200	100
A:=A+100			
Write(A)		300	
Read(B)			
B:=B+100			
Write(B)			200
	Read(A)		
	A:=A*2		
	Write(A)	600	
	Read(B)		
	B:=B*2		
	Write(B)		400

T1	T2	A	B
	Read(A)	200	100
	A:=A*2		
	Write(A)	400	
	Read(B)		
	B:=B*2		
	Write(B)		200
Read(A)			
A:=A+100			
Write(A)		500	
Read(B)			
B:=B+100			
Write(B)			300

- 结果不一样是否意味结果不正确？
  - 如果一个事务运行时，没有其他事务同时运行，或者说没有受到其他事务的干扰，那么结果是正确的。
- 多个事务串行执行的结果是正确的。
- 多个事务交叉执行（并发）结果是否正确？
  - 有的正确，有的不正确。
  - 如何判别执行结果是否正确？
    ——与某个串行执行结果相同。



- 串行调度
  - 一个事务的第一个操作是在另一个事务的最后一个操作完成后开始。即调度中事务的各个操作不会交叉，每个事务相继执行。
  - 串行调度总是可以正确执行。
  - 串行调度效率很低。
- 可串行化调度
  - 如果一个并发调度的结果与某个串行调度的结果相同，则该调度称为可串行化调度。



- 可串行化是并发事务正确调度的准则。
- 多个事务的并发执行是正确的，当且仅当并发执行的结果与这些事务按某一串行顺序执行的结果相同。
- **DBMS**一般提供必要的措施（并发控制技术）保证多个事务的并发调度是可串行化的。

可串行化调度

T1	T2	A	B
Read(A)		200	100
A:=A+100			
Write(A)		300	
	Read(A)		
	A:=A*2		
	Write(A)	600	
Read(B)			
B:=B+100			
Write(B)			200
	Read(B)		
	B:=B*2		
	Write(B)		400

不可串行化调度

T1	T2	A	B
Read(A)		200	100
A:=A+100			
Write(A)		300	
	Read(A)		
	A:=A*2		
	Write(A)	600	
	Read(B)		
	B:=B*2		
	Write(B)		200
Read(B)			
B:=B+100			
Write(B)			300

调度的冲突等价性

- 冲突操作
  - 不同事务对同一数据项进行的两个操作中，有一个写操作，两者即为冲突操作。
  - 调度的操作之间可能存在冲突
    - 读-写冲突， $R_i(x)$与$W_j(x)$
    - 写-写冲突， $W_i(x)$与$W_j(x)$
  - 同一事务的操作之间没用冲突的概念。
  - 不同数据项的操作之间没有冲突的概念。
- 冲突操作的操作顺序，会影响操作的结果。
- 非冲突操作的操作顺序，不影响操作结果。

T1	T2
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

T1	T2
Read(A)	
Write(A)	
	Read(A)
Read(B)	
	Write(A)
Write(B)	
	Read(B)
	Write(B)

T1	T2
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

- 调度的冲突等价性
  - 不同调度S_1和S_2是等价的，如果：对任意一对冲突操作$< O_i, O_j >$，在调度S_1中O_i优先O_j而在调度S_2中O_i也优先O_j。
  - 如果一个调度S能通过一系列非冲突操作执行顺序的交换变成调度S1，则称调度S和S1冲突等价。
  - 如果一个调度冲突等价于某个串行调度，则该调度称为冲突可串行化的调度。
  - 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。

调度的状态等价性

- 冲突等价性过于严格，排斥了很多正确的并发调度。调度的目的是保持数据库状态的一致性。
- 如果初始数据库状态相同，两个不同的调度能产生相同的数据库状态，则认为这两个调度是状态等价的。这种等价性称为数据库状态等价性。
- 一个调度是状态可串行的，如果它状态等价于一个串行调度。
- 一个调度是冲突可串行的，一定是状态可串行的。

- 两个调度S_1和S_2状态等价应满足的条件

  1）对每个数据项A，如果S_1中事务T_1读A初值，则S_2中的T_1也必须读A初值

  2）如果S_1中事务T_1执行Read（A），则S_2中T_1也必须执行Read(A)，并且，如果S_1中T_1读到的A值是由事务T_2产生的，则S_2中T_1读到的A值也是由T_2产生的。

  3）如果S_1中有一个事务执行最后的Write（A）操作，则该事务在S_2中也必须执行最后的Write（A）操作。

  - 两个调度中同一个事务所读取的同一数据项的值都是相同的；
  - 同一数据项在不同调度中的最后写操作，必须由同一事务执行（保证最终的数据库状态一致）。

- 状态可串行调度示例：
  - 3个事务：
    T1=W1(Y)W1(X),T2=W2(Y)W2(X),T3=W3(X)
  - 2个调度：
    S1={W1(Y)W1(X)W2(Y)W2(X)W3(X)}
    S2={W1(Y)W2(Y)W2(X)W1(X)W3(X)}
  - S1是一个串行调度。
  - S2不满足冲突可串行化，因为通过不能交换非冲突操作变成串行调度。
  - S2是可串行化的，因为S2执行的结果与调度S1相同，Y的值都等于T2的值，X的值都等于T3的值。

调度的可串行性测试

- 基于冲突等价的可串行性测试方法
  - 构造调度的优先图（前趋图）

    调度 S 的优先图是一个有向图G(V,E)，其中

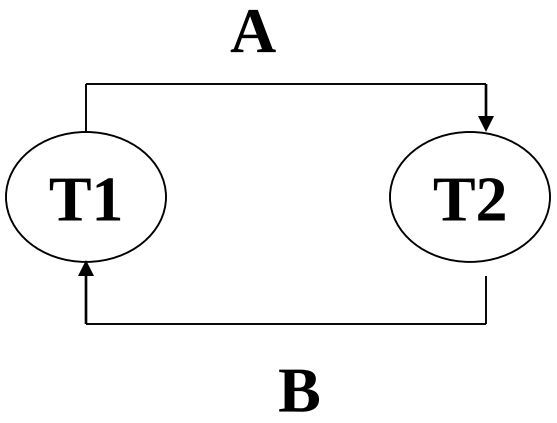
    V: 一组节点$V=\{T_1T_2,\ldots,T_n\}$， S中的事务

    E: 一组有向边$E=\{e_1,e_2,\ldots,e_n\}$，

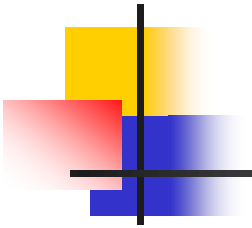
    $T_i \rightarrow T_j$ 是图中的一条边，当且仅当$\exists\ p \in T_i, q \in T_j$ 使得p, q 冲突，并且 $p <_S q$（p在q的前面执行）
  - 当且仅当优先图中没有回路时，调度S才是可串行化的。

T1	T2	A	B
Read(A)		200	100
A:=A+100			
Write(A)		300	
	Read(A)		
	A:=A*2		
	Write(A)	600	
	Read(B)		
	B:=B*2		
	Write(B)		200
Read(B)			
B:=B+100			
Write(B)			300

不可串行化调度



调度优先图



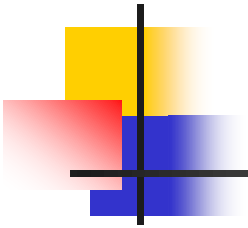
- 如何保证并发操作的调度是正确的
  - 为了保证并发操作的正确性，**DBMS**的并发控制机制必须提供一定的手段来保证调度是可串行化的。
- 常用的并发控制技术有
  - 封锁**(Locking)**
  - 时间戳**(Timestamp)**
  - …
- 商用的**DBMS**一般都采用封锁方法

9.3 基于封锁的并发控制技术

- 基本思想：当事务T_1要修改记录A，如果修改A之前先给A加锁，使得T_2和其它事务不能读取和修改A，直到T_1修改并写回A后解除对A的封锁为止。
- 结果：既不会丢失事务T_1的更新，也不会出现读“脏”数据的问题。
- 锁

  ——数据项上的并发控制标志
- 封锁

  ——事务对数据加上锁



- 封锁是实现并发控制的一个非常重要的技术
  - 事务T在对某个数据对象(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。
- 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。

- 锁的基本类型
  - 共享锁（**Share lock**，简记为**S**锁，又称为读锁）
    - 若事务**T**对数据对象**A**加上**S**锁，则其它事务只能再对**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。
  - 排它锁（**eXclusive lock**，简记为**X**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。

- 锁的使用
  - 事务在对数据对象进行操作前，需申请相应类型的锁；
  - 只有获得所需锁后，事务才能继续其操作；
  - 能否获得锁是由锁的相容性决定的。
- 锁的相容性矩阵

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

X：排他锁

S：共享锁

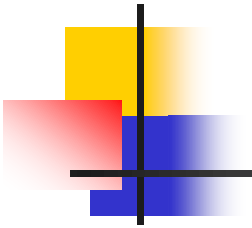
—：无锁

Y：相容

N：不相容

- 列表示事务**T1**已经获得的数据对象上的锁的类型，其中横线表示没有加锁；
- 行表示事务**T2**对同一数据对象发出的封锁请求；
- **T2**的封锁请求能否被满足用矩阵中的**Y**和**N**表示；
- **Y**表示事务**T2**的封锁要求与**T1**已持有的锁相容，封锁请求可以满足；
- **N**表示**T2**的封锁请求与**T1**已持有的锁冲突，**T2**的请求被拒绝。

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



- 锁的相容性表明
  - 一个数据项可以同时有多个共享锁；
  - 一个数据项上只允许一个排它锁；
  - 排它锁请求必须等到该数据项上的所有共享锁或排它锁被释放后才有效。
- 锁的正确使用要保证
  - 事务的一致性
    - 先封锁，后读写
    - 封锁与解锁必须匹配
  - 调度的合法性
    - 不能采用不相容的锁同时封锁同一数据项

封锁协议

- 封锁协议（加锁的规则）
  - 在给数据对象加锁时，要考虑使用何种类型的锁、何时请求锁、持有锁的时间和何时释放等，要遵从一定规则。这些规则被称为封锁协议。
  - 按照锁的类型和释放锁的时机，可以将封锁协议分为
    - 一级封锁协议
    - 二级封锁协议
    - 三级封锁协议

- 一级封锁协议
  - 事务T在修改数据A前必须先对其加X锁，直到事务结束才释放
  - 没有丢失更新
  - 读数据不加锁，所以它不能保证可重复读和不读“脏”数据

T_1	T_2
①Xlock A 获得	
②读A=16	
	Xlock A
③A←A-1	等待
写回A=15	等待
Commit	等待
Unlock A	等待
④	获得Xlock A
	读A=15
	A←A-1
⑤	写回A=14
	Commit
	Unlock A

T_1	T_2
① Xlock A 获得 ② 读A=16 A←A-1 写回A=15 ③ ④ Rollback Unlock A	读A=15

- 读“脏”数据
- 不可重复读

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

- 二级封锁协议
  - 在一级封锁协议基础上，规定：事务T在读数据A之前必须先对其加S锁，读完后即可释放S锁
  - 二级封锁协议的目的是：防止读“脏”数据。

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

- 二级封锁协议并不能避免不可重复读的问题。
  - 因为事务T在读数据A之前加上的S锁，读完后即释放了，以后再读时有可能数据发生了变化。

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

不可重复读 ⟶

- 三级封锁协议：
  - 在二级封锁协议基础上，规定：某一事务施加的**S**锁要保持到该事务结束时才释放。
  - 效率不高

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

- 不同级别封锁协议的对比
  - 区别：什么操作需要申请封锁以及何时释放锁；
  - 作用：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	不丢失更新	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		√		√	√	√	√

活锁与死锁

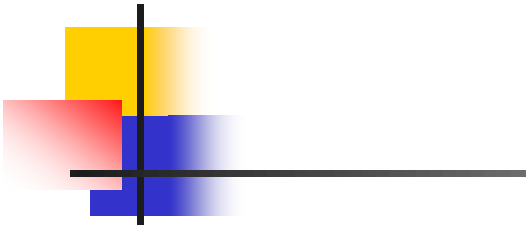
- 封锁技术可以有效地解决并发操作的一致性问题，但也带来一些新的问题：死锁，活锁
  - 活锁：在数据库系统中活锁是指某个事务由于请求封锁，但总也得不到锁而长时间处于等待状态。
  - 死锁：是指同时处于等待状态的两上或多个事务相互封锁了对方请求的资源，使得没有任何一个事务可以获得足够的资源运行完毕，而永远等待下去。

T_1	T_2	T_3	T_4
Lock A Unlock Wait Wait Wait Wait	Lock A wait wait Wait Wait Wait Wait Wait Wait	Lock A Wait Lock A Unlock	Lock A Wait Wait Lock A

活锁

T_1	T_2
Lock A	Lock B
Lock B Wait Wait Wait Wait	Lock A Wait Wait

死锁

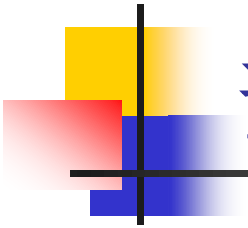


- 如何避免活锁
  - 采用先来先服务的策略：
    - 当多个事务请求封锁同一数据对象时，按请求封锁的先后次序对这些事务排队；
    - 该数据对象上的锁一旦释放，首先批准申请队列中第一个事务获得锁。
- 死锁如何处理
  - **1.** 预防死锁；
  - **2.** 检测死锁，然后解除死锁。

- 产生死锁的原因
  - 两个或多个事务都已封锁了一些数据对象，然后又都请求被其他事务封锁的数据对象，从而出现互相等待的死锁状态。

T1	T2
Xlock R_1	.
.	.
.	Xlock R_2
.	.
Xlock R_2	.
等待	Xlock R_1
等待	等待
等待	等待
.	.

各自部分封锁+互相等待



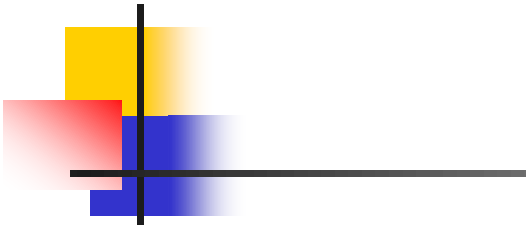
预防死锁

- 预防死锁
  - 破坏产生死锁的条件
  - 方法
    - 一次封锁法
    - 顺序封锁法
    - 事务重试法

- 一次封锁法
  - 要求每个事务在开始执行之前，获得所有数据项上的锁
  - 不存在持锁等待
  - 不会发生死锁
  - 一次封锁法存在的问题
    - 降低了系统的并发度，因为将以后要用到的全部数据加锁，势必扩大了封锁的范围；
    - 很难事先精确地确定每个事务所要封锁的数据对象。

- 顺序封锁法
  - 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。
  - 不会出现循环等待，从而避免死锁。
  - 顺序封锁法存在的问题
    - 数据库系统中可封锁的数据对象极其众多，并且随数据的插入、删除等操作而不断地变化，要维护这样极多而且变化的资源的封锁顺序非常困难，成本很高。 维护成本高
    - 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。难于实现

      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。



- 事务重试法
  - 使用抢占机制和事务回滚；
  - 当事务**T2**申请的锁已被事务**T1**占有时，根据事务开始的先后，授予**T1**的锁可以通过回滚事务**T1**而被抢占，将**T1**释放的锁授予**T2**，而事务**T1**回滚后自动重试。

死锁的检测与恢复

- 死锁预防的方法可以避免死锁，但是较难实现，效率也比较低，不太适合数据库的特点。
- **DBMS**处理死锁问题的一般策略是：
  - 允许死锁发生；
  - 由**DBMS**的并发控制子系统定期检测系统中是否存在死锁（检测死锁）；
  - 一旦检测到死锁，设法解除死锁：
    - 回滚一个或多个相关事务，释放其所持有的锁，使其他事务能够继续运行下去；
    - 一般选择回滚代价最小的事务进行回滚。

- 死锁检测方法
  - 超时法
  - 事务等待图法
- 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：可能误判死锁：
    - 时限太长，死锁发生后不能及时发现，导致不必要的延误；
    - 时限太短，即使没有死锁也可能引起事务回滚。

- 事务等待图法
  - 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=(V，U)**
    - **V**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**V1**等待**V2**，则**V1**，**V2**之间画一条有向边，从**V1**指向**V2**。
  - 并发控制子系统周期性地检测事务等待图，如果发现图中存在回路，则表示系统中出现了死锁。

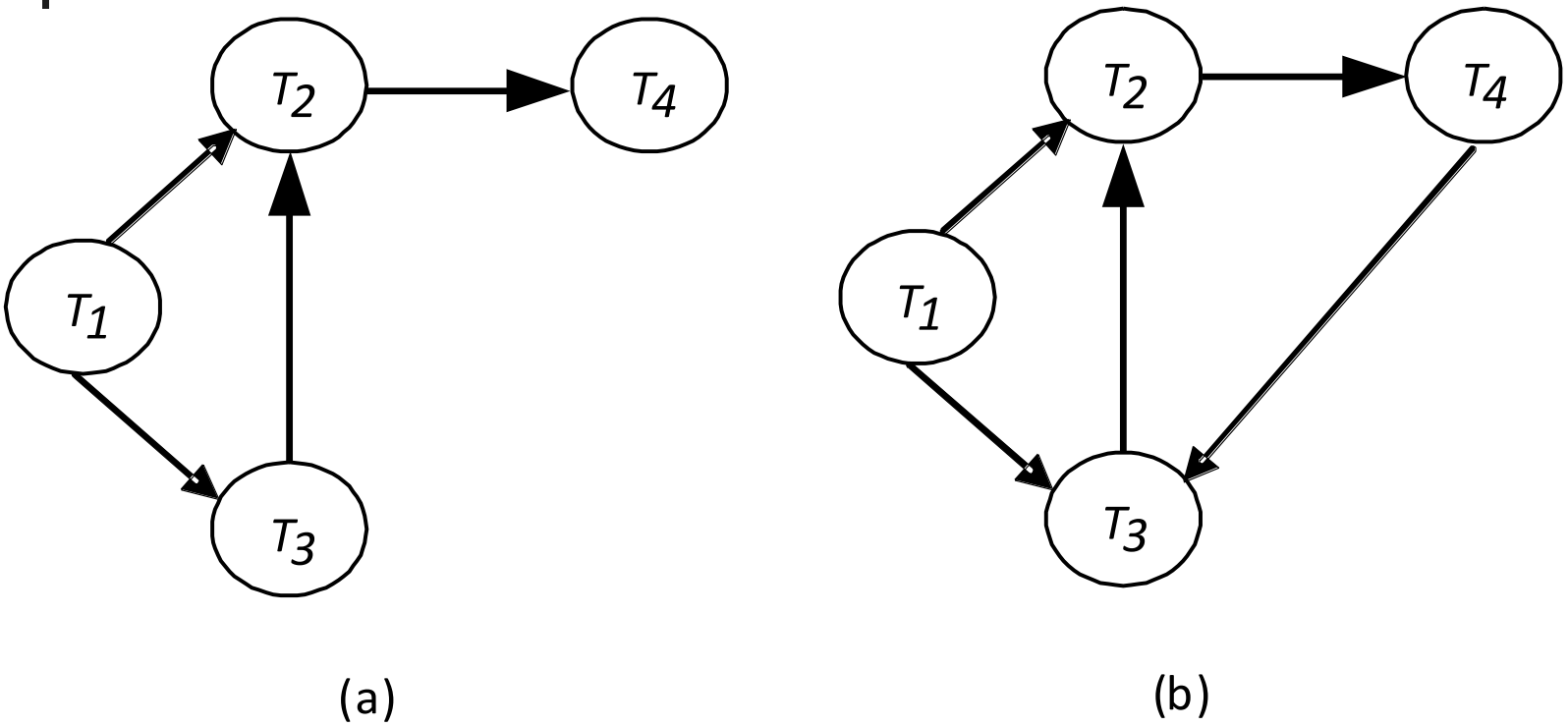


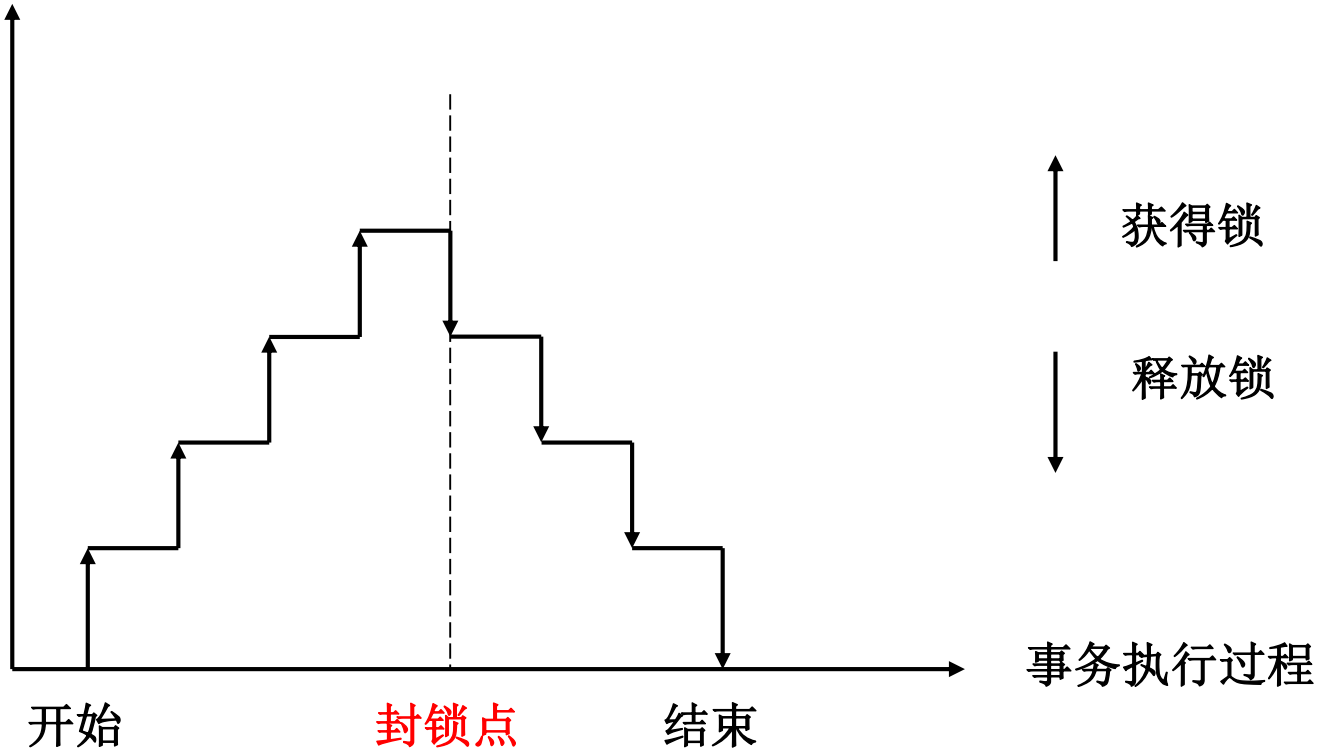
图9-20　事务等待图。（a）无环等待图。（b）有环等待图

- 死锁的恢复
  - 解除死锁的方法是回滚一个或多个相关事务
    - 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锁，使其它事务能继续运行下去。
    - 回滚代价的计算可以包括：事务计算时间的长短、使用数据项的多少、回滚时牵涉多少事务，等等。

两阶段封锁协议

- 封锁协议
  - 对数据对象加锁时约定的一些规则
    - 锁的类型
    - 何时申请封锁
    - 持锁时间
    - 何时释放封锁
  - 一级封锁协议
  - 二级封锁协议
  - 三级封锁协议
  - 两阶段封锁协议**(Two-Phase Locking，2PL)**

- 两阶段封锁协议
  - 常用的一种封锁协议
  - 要求：
    - 任何事务在对数据进行操作前必须先获得锁；
    - 一个事务所有的封锁操作都在第一个解锁操作之前。
  - 事务的执行分为两个阶段：
    - 第一阶段　获得锁阶段，也称为扩张阶段。在这阶段，事务可以申请获得任何数据项上任何类型的锁，也可以进行锁的升级转换，但是不能释放任何锁；
    - 第二阶段　释放锁阶段，也称为收缩阶段。在这阶段，事务可以释放任何数据项上的任何类型的锁，也可以进行锁的降级转换，但是不能再申请任何锁。



两阶段封锁协议

- 两阶段封锁协议
  - 理论上可以证明：使用两段封锁协议产生的调度是可串行化调度
  - 并行执行的所有事务都遵守两阶段封锁协议，则对这些事务的所有并行调度都是可串行化的。
  - 所有遵守两阶段封锁协议的事务，其并行执行的结果也一定是正确的。
  - 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。
  - 在可串行化的调度中，不一定所有事务都必须符合两阶段封锁协议。

T_1	T_2
Slock B	
读B=2	
Y=B	
Xlock A	
	Slock A
A=Y+1	等待
写回A=3	等待
Unlock B	等待
Unlock A	等待
	Slock A
	读A=3
	Y=A
	Xlock B
	B=Y+1
	写回B=4
	Unlock B
	Unlock A

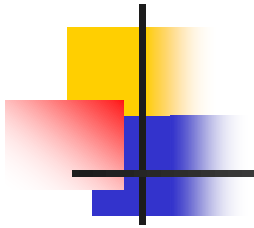
T1和T2遵循2PL

T_1	T_2
Slock B	
读B=2	
Y=B	
Unlock B	
Xlock A	
	Slock A
A=Y+1	等待
写回A=3	等待
Unlock A	等待
	Slock A
	读A=3
	X=A
	Unlock A
	Xlock B
	B=X+1
	写回B=4
	Unlock B

T1和T2不遵循2PL

- 两阶段封锁协议不能避免死锁
  因为并不要求事务必须一次将所有要使用的数据全部加锁

T_1	T_2
Slock B	
读B=2	
	Slock A
	读A=2
Xlock A	
等待	Xlock B
等待	等待



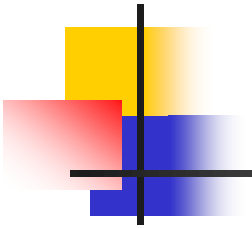
- 为了避免死锁，采用一次封锁法，将**2PL**改成保守的**2PL**
  - 事务在操作执行前获得所有操作数据上的锁。
  - 一次封锁所有数据项，否则等待。
  - 不会产生死锁，但难以实现，因为需要预先知道事务所需要处理的数据。

锁的管理—锁表

- 如何维护锁
  - 锁的请求、授予和解除是由数据库系统的锁管理器**(Lock Manager)** 维护。
  - 锁管理器为目前已加锁的数据项维护一个记录链表，这个链表称为锁表（**Lock Table**）。
  - 一条锁表记录表示一个锁请求，内容包括：请求锁的事务、请求的锁类型、是否已经授予锁等
  - 所有请求按照到达的先后顺序排序

哪些数据项加了锁，加了什么锁，

谁加的锁，谁正在封锁，谁在等待封锁



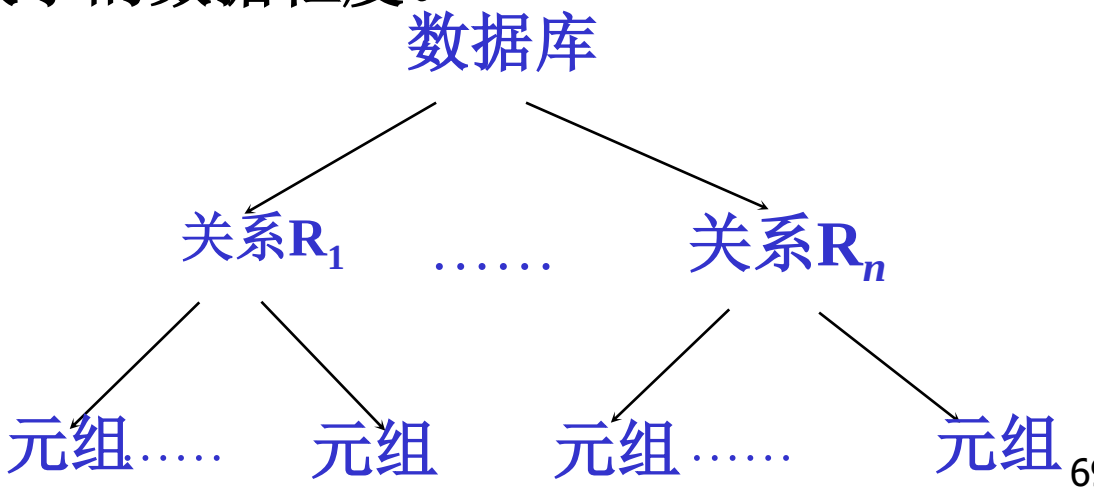
- 锁表是将数据库元素和封锁信息联系在一起的一个关系，记录授予的锁和挂起的请求。
- 封锁请求处理
  - 假设事务**T**请求数据库元素**A**上的锁，如果没有**A**的锁表项，则表明**A**无锁，因此相应的表项被创建。
  - 如果存在**A**的锁表项，新的请求将加入到请求队列末尾，并且如果与所有以前的锁兼容则被授予。
- 释放锁的处理
  - 释放锁将导致锁表中相关的请求项被删除，并检查其后的封锁请求是否可以被授予。

9.4 多粒度封锁

- 封锁粒度
  - 封锁对象可以是数据库的逻辑单元，也可以是物理单元。
    - 逻辑单元
      属性列、元组、关系（表）、索引项、整个索引、整个数据库，等等。
    - 物理单元
      页（数据页或索引页）、物理记录、数据库存储空间等。
  - 不同数据对象的大小不一样。
  - 封锁对象的大小称为封锁粒度。

- 如何选择封锁粒度
  - 选择封锁粒度需要综合考虑系统开销和并发度两个因素
    - 封锁粒度越小，并发度越高，需要的锁越多，系统开销大。
    - 封锁粒度越大，并发度越低，需要的锁就少，系统开销小。
  - 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：以关系为封锁单位。
  - 只处理少量元组的用户事务：以元组为封锁单位。

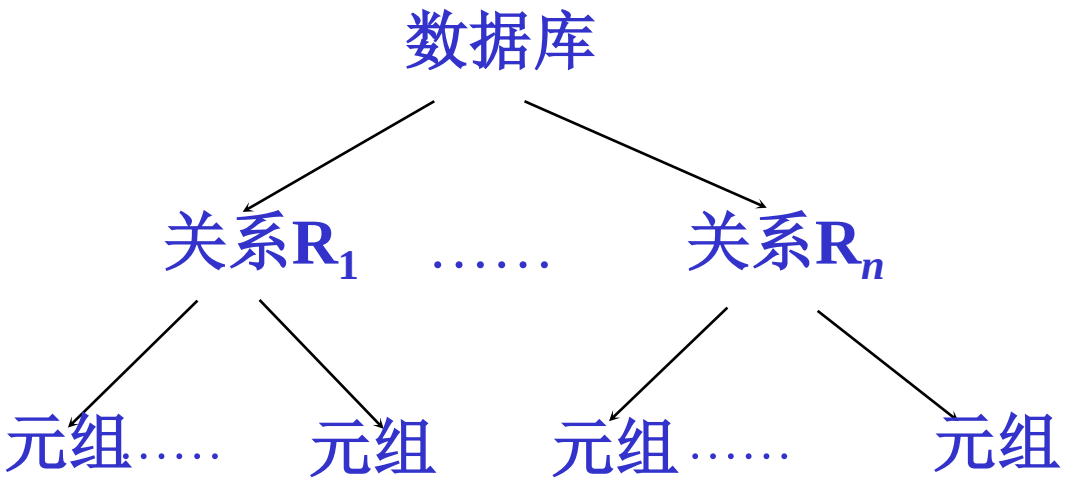
- 多粒度封锁机制
  - 在一个系统中同时支持多种封锁粒度；
  - 允许事务选择不同大小的粒度作为封锁单元；
  - 不同的封锁粒度可以用多粒度树表示
    - 以树形结构来表示多级封锁粒度；
    - 根结点是整个数据库，表示最大的数据粒度；
    - 叶结点表示最小的数据粒度。



- 多粒度封锁需要执行多粒度封锁协议
  - 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（封锁的效果是一样的）：
    - 显式封锁
      ——直接加到数据对象上的封锁。
    - 隐式封锁
      ——由于其上级结点加锁而使该数据对象加上了锁。

- 系统检查封锁冲突时
  - 既要检查显式封锁，还要检查隐式封锁
  - 对某个数据对象加锁时，系统检查的内容包括
    - 1）该数据对象本身有无显式封锁冲突
    - 2）是否与所有上级结点的显式封锁冲突（上级结点的显式封锁相当于本结点的隐式封锁）
    - 3）是否与所有下级结点的显式封锁冲突（本结点的显式封锁将成为下级结点的隐式封锁）
  - 检查的内容多，效率低

- 例：事务T要对关系R1加X锁
  - 系统必须搜索其上级结点数据库；
  - 还要搜索R1的下级结点，即R1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。

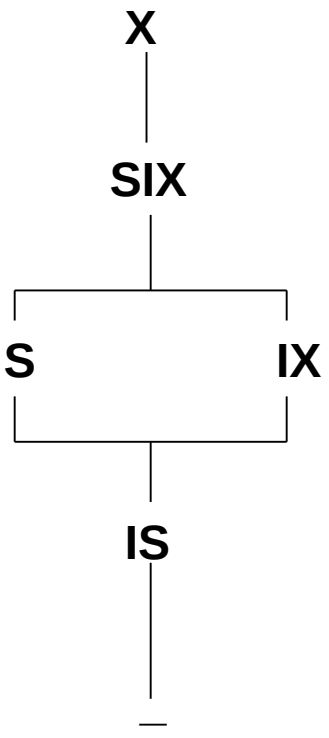


- 为了提高多粒度封锁的检查效率，在多粒度封锁机制中引入了意向锁（**intention lock** ）
- 什么是意向锁
  - 对任一结点加基本锁（普通锁），
    必须先对它的上层结点加意向锁；
  - 如果对一个结点加意向锁，
    则说明该结点的某个下层结点正在被加锁（普通锁）。

- 常用意向锁
  - 意向共享锁(Intent Share Lock，简称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:要对某个元组加S锁，则要先对关系和数据库加IS锁
  - 意向排它锁(Intent Exclusive Lock，简称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:要对某个元组加X锁，先要对关系和数据库加IX锁
  - 共享意向排它锁(Share Intent Exclusive Lock, 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）

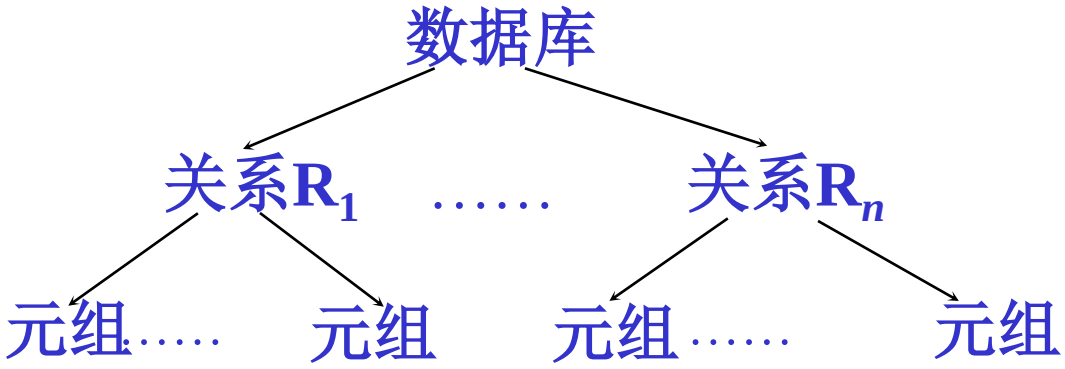
- 增加了意向锁的相容矩阵

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**



锁的强弱关系

- 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：事务**T1**要对关系**R1**加**S**锁
    - 要首先对数据库加**IS**锁；
    - 检查数据库和**R1**是否已加了不相容的锁**(X**或**IX)**；
    - 不再需要搜索和检查**R1**中的元组是否加了不相容的锁**(X**锁**)** 。

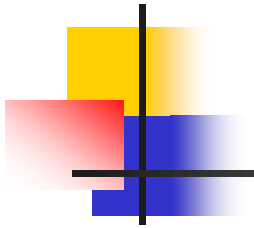


- 意向锁的小结
  - 具有意向锁的多粒度加锁方法中，任意事务T要对一个数据对象加锁，必须先对它的上层节点加意向锁；
  - 申请封锁时应该按自上而下的次序进行；
  - 释放锁时则应该按自下而上的次序进行；
  - 具有意向锁的多粒度加锁方法提高了系统的并发度，减少了加锁和释放锁的开销；
  - 在实际的数据库管理系统产品中得到广泛应用。

SIX和IS相容？　　SIX和IX相容？

9.8 SQL Server的并发控制

- **SQL Server**对事务的存取模式和隔离级别作了具体规定，提供给用户选择。
- **SQL Server**的并发控制基于封锁技术，系统支持三级封锁协议，采用了多粒度封锁策略。
- 除了提供**S**锁和**X**锁这些基本锁外，**SQL Server**也提供了意向锁，以及一些专用锁。



- **SQL Server的存取模式**
  两种基本的事务存取模式
  - **READ ONLY**（只读型事务），对数据库的操作只能是读操作；
  - **READAND AND WRITE**（读写型事务），对数据库可以是读操作，也可以是写操作。

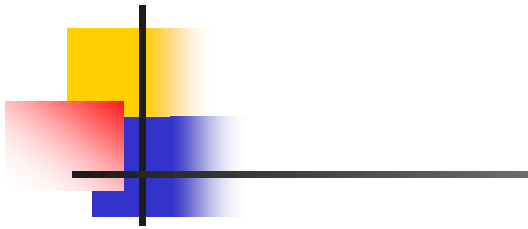
- 事务的隔离级别
  - 尽管可串行性对于事务确保数据库中的数据在所有时间内的正确性相当重要，然而许多事务并不总是要求完全的隔离
  - 事务准备接受不一致数据的级别称为隔离级别。
  - 隔离级别是一个事务必须与其它事务进行隔离的程度。
    - 较低的隔离级别可以增加并发，但是会降低数据的正确性。
    - 较高的隔离级别可以确保数据的正确性，但可能对并发产生负面影响。
  - 应用程序要求的隔离级别确定了 **SQL Server** 使用的锁定行为。

- **SQL Server支持SQL-92的四个隔离级别**
  - **1）SERIALIZABLE**，可串行化：允许事务与其他事务并发执行，但系统必须保证并发调度是可串行化，不会导致发生错误。
  - **2）REPEATABLE READ**，可重复读：只允许事务读已提交的数据，并且在两次读同一数据时不允许其他事务修改此数据。
  - **3）READ COMMITTED**，读提交数据：允许事务读已提交的数据，但不要求“可重复读”。在事务对同一记录的两次读取之间，记录可能被已提交的事务更新。
  - **4）READ UNCOMMITTED**，读未提交的数据：允许事务读已提交或未提交的数据。

- 不同隔离级别的比较

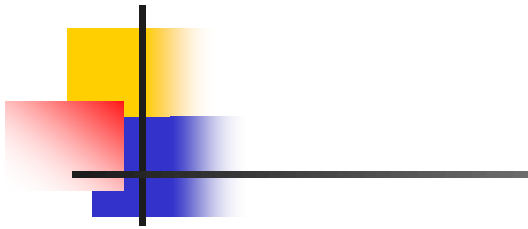
选项	描述
READ UNCOMMITTED	未提交读。使用写锁、不使用共享锁，允许读脏数据。
READ COMMITTED	提交读。在读取时使用共享锁，不允许读脏数据。
REPEATABLE READ	可重复读。保持读锁直到事务结束，不存在读脏数据和不可重复读取问题。
SERIALIZABLE	可串行化。防止其他事务更新或插入符合本事务中 **WHERE**子句条件的行。结果正确，不可能产生幻读。

- 上述4种级别可以用下列SQL语句定义：
  - **SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | READ UNCOMMITTED | REPEATABLE READ | SERIALIZABLE };**
  - 默认为**READ COMMITTED**
  - 较低的隔离级别可以增加并发性，但代价是降低数据的正确性；
  - 相反，较高的隔离级别可以确保数据的正确性，但可能对并发产生负面影响。
  - 事务必须运行于可重复读或更高的隔离级别以防止丢失更新。

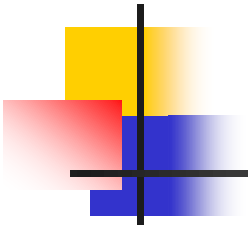


- **SQL Server的锁**
  - 基本锁（包括共享锁和排它锁）
    - 共享锁：用于数据的只读操作，如 **SELECT** 语句
    - 排它锁：用于数据的修改操作，例如 **INSERT**、**UPDATE** 或 **DELETE**
  - 意向锁（ **SQL Server** 内部使用意向锁）
  - 专用锁
    - 更新锁（**Update Lock**）
    - 架构锁（**Scheme Lock**）
    - 大容量更新锁

- 更新锁
  ——可以避免更新事务出现死锁
  - 更新事务需要先获取S锁读取记录，然后将S锁转化为X锁，修改记录；
  - 如果两个事务都获得了该数据上的**S**锁，且试图同时更新数据，那么一个事务会尝试将锁转化为**X**锁。转化需要等待一段时间。第**2**个事务试图获取**X**锁以进行更新。此时，发生了死锁。
  - 一次只有一个事务可以获得数据的更新锁。如果事务修改数据，则更新锁转化为排它锁。否则，转换为共享锁



- 架构锁（**Scheme Lock**）
  - 包括：架构修改 **(Sch-M)** 锁和架构稳定性 **(Sch-S)** 锁。
  - 提供了对数据定义操作的控制。
  - 在执行依赖于表架构的操作时使用，确保表或索引在被另外的会话引用时不被删除或更改架构。
  - 编译查询时，使用**Sch-S**，确保数据对象不被删除。
  - 执行**DDL**操作时，使用**Sch-M**，实现表的架构的修改。



- 大容量更新锁
  - 作用：允许将大容量的数据并发地复制到同一表中，同时防止其他不进行大容量数据复制的事务访问该表。
  - 当将数据大容量复制到表，且指定了 **TABLOCK** 提示或者使用 **sp_tableoption** 设置了 **table lock on bulk** 选项时，将使用大容量更新锁。

- 锁的使用和管理
  - 在**SQL Server**中，使用**SELECT**、**INSERT**、**UPDATE**和**DELETE**语句可以指定表级的锁定类型。
  - 设置共享锁

    SELECT OrderID, CustomerID, OrderDate FROM Orders WITH (HOLDLOCK)

    -- 在检索订单报表Orders时，用WITH (HOLDLOCK)设置共享锁
  - 设置排它锁

    INSERT INTO Orders WITH (TABLOCKX)(CustomerId, OrderDate) VALUES ('ALFKI', '2002-01-01')

- **SQL Server**提供了一个系统存储过程**sp_lock**，用来获取有关锁的信息。
- **SQL Server**定期执行死锁检测，能自动发现并解除死锁。解除死锁的方法是：撤销（回滚）更新最少的事务，或者是开始较晚的事务。
- **SQL Server**提供超时功能，如果事务锁定某个资源超时了，系统将自动回滚该事务，释放资源，有效地避免事务阻塞。

本章小结

- 并发操作带来的数据不一致性
  - 丢失更新（**lost update**）
  - 不可重复读**(non-repeatable read)**
  - 读“脏”数据**(dirty read)**
  - 幻读**(phantom read)**
- 并发控制机制的任务：
  - 保证事务的隔离性
  - 保证数据库的一致性
- 可串行化调度的定义
  - 可串行化是并发事务正确调度的准则
  - 可串行化调度的充分条件：冲突可串行化

- 数据库的并发控制以事务为单位、使用封锁机制
  - 锁的类型：**S**、**X**
  - 封锁协议：
    一级封锁协议，二级封锁协议，三级封锁协议
- 对数据对象施加封锁，带来问题
  - 活锁：先来先服务
  - 死锁：预防、检测和解除
- 两阶段封锁协议
- 多粒度封锁
  - 意向锁（**IS**、**IX**、**SIX**）

- 并发控制机制调度并发事务是否正确的判别准则是可串行性
  - 并发操作的正确性通常由两阶段封锁协议来保证.
  - 两阶段封锁协议是可串行化调度的充分条件，不是必要条件.
- 不同的数据库管理系统提供的封锁类型、封锁协议、达到的系统一致性级别不尽相同，但其依据的基本原理和技术是相通的。

作业和思考题

- 作业
  P206（e216）：8，13

- 思考题
  P206－207：1，2，3，4，5，6，7，9，10