并发控制

单世民

并发执行

- 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
  - (2)交叉并发方式（interleaved concurrency）
    - 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
    - 是单处理机系统中的并发方式，能够减少处理机的空闲时间，提高系统的效率
  - (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

并发执行

- 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必须提供并发控制机制
  - 并发控制机制是衡量一个DBMS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

并发控制

- 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

并发操作的潜在不一致性

- 丢失修改（lost update）
- 不可重复读（non-repeatable read）
- 读“脏”数据（dirty read）

并发操作的潜在不一致性

- 丢失修改

  丢失修改是指事务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。

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

Ssm

并发操作的潜在不一致性

- 读“脏”数据
  事务1修改某一数据，并将其写回磁盘。事务2读取同一数据后，事务1由于某种原因被撤消，这时事务1已修改过的数据恢复原值，事务2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。

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	

Ssm

并发操作的潜在不一致性

- 不可重复读
  不可重复读是指事务1读取数据后，事务2执行更新操作，使事务1无法再现前一次读取结果。

- 三类不可重复读
  事务1读取某一数据后：
  - 1. 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。
  - 2. 事务2**删除**了其中部分记录，当事务1再次读取数据时，发现某些记录神秘地**消失**了。
  - 3. 事务2**插入**了一些记录，当事务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）

并发操作的潜在不一致性

- 不可重复读

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)	

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

- DBMS通常提供了多种类型的封锁。一个事务对某个数据对象加锁后究竟拥有什么样的控制是由封锁的类型决定的。

基本封锁类型

- 排他锁（eXclusive lock，简记为X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
- 共享锁（Share lock，简记为S锁）
  - 共享锁又称为读锁
  - 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

基于锁的协议

- 每个事务都要根据自己将对数据项进行的操作类型申请适当的锁。只有在事务获得所需锁后，事务才能继续其操作。

	S	X
S	true	false
X	false	false

锁相容性矩阵

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：封锁协议（Locking Protocol）
  - 何时申请X锁或S锁
  - 持锁时间、何时释放
- 不同的封锁协议，在不同的程度上为并发操作的正确调度提供一定的保证
- 常用的封锁协议：3级封锁协议

1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）
- 1级封锁协议可防止丢失修改
- 在1级封锁协议中，如果是读数据，不需要加锁的，所以它不能保证可重复读和不读“脏”数据。

1级封锁协议

T_1	T_2
① Xlock 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

没有丢失修改

Ssm

1级封锁协议

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

读“脏”数据

1级封锁协议

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
	Commit
	Unlock B
③读A=50	
读B=200	
求和=250	
(验算不对)	

不可重复读

2级封锁协议

- 1级封锁协议+**事务T在读取数据R前必须先加S锁，读完后即可释放S锁**
- 2级封锁协议可以防止丢失修改和读“脏”数据。
- 在2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。

2级封锁协议

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

T_1 (续)	T_2
④Slock A	
获得	
读A=50	
Unlock A	
Slock B	
获得	
读B=200	
Unlock B	
求和=250	
(验算不对)	

不可重复读

3级封锁协议

- 1级封锁协议 + **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**

- 3级封锁协议可防止丢失修改、读脏数据和不可重复读。

3级封锁协议

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

可重复读

Ssm

3级封锁协议

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

不读“脏”数据

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
1级封锁协议		√			√		
2级封锁协议		√	√		√	√	
3级封锁协议		√		√	√	√	√

- 三级协议的主要区别
  - ¤ 什么操作需要申请封锁
  - ¤ 何时释放锁（即持锁时间）

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

活锁

- 一个事务总是不能在某数据项上加上锁，因此该事务也就永远不能取得进展。（锁的分配不公平）
- 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

- 死锁
- 如：由于事务占有锁并且申请其它的锁，使得这些事务不能继续推进的这种情况。

- 解决死锁的方法：两类
  - ¤ 1. 预防死锁
  - ¤ 2. 死锁的诊断与解除

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

- 预防死锁的方法
  - **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，否则不能继续执行。（预知难、等待时间长、降低并发度）
  - **顺序封锁**预先对数据对象规定一个封锁顺序，所有的事务都按这个顺序实行封锁。

死锁的预防

- **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

- 一次封锁法存在的问题：降低并发度
  - ¤ 扩大封锁范围，将以后要用到的全部数据加锁，势必扩大了封锁的范围，从而降低了系统的并发度
  - ¤ 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。

死锁的预防

- **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
  - 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

- **结论**
  - 在操作系统中广为采用的预防死锁的策略并不很适合数据库的特点
  - **DBMS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**

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

死锁的诊断与解除

- **检测死锁：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

死锁的诊断与解除

- **检测死锁：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T1等待T2，则T1，T2之间划一条有向边，从T1指向T2
- 并发控制子系统周期性地（比如每隔1 min）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**。

死锁的诊断与解除

- **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。

正确的并发调度

- 什么样的并发操作调度是正确的
- 如何保证并发操作的调度是正确的

什么样的并发操作调度是正确的

- 计算机系统对并行事务中并行操作的调度是随机的，而不同的调度可能会产生不同的结果。

- **将所有事务串行起来的调度策略一定是正确的调度策略。**
  - 如果一个事务运行过程中没有其他事务在同时运行，也就是说它没有受到其他事务的干扰，那么就可以认为该事务的运行结果是正常的或者预想的
  - 以不同的顺序串行执行事务也有可能会产生不同的结果，但由于不会将数据库置于不一致状态，所以都可以认为是正确的。
  - 几个事务的并行执行是正确的，**当且仅当其结果与按某一次序串行地执行它们时的结果相同**。这种并行调度策略称为**可串行化**（Serializable）的调度。

什么样的并发操作调度是正确的

- **可串行性是并行事务正确性的唯一准则**

- 例：现在有两个事务，分别包含下列操作：
  事务1：读B；A=B+1；写回A；
  事务2：读A；B=A+1；写回B；
  假设A的初值为2，B的初值为2。
- 对这两个事务的不同调度策略
  - 串行执行
    - 串行调度策略1（S1）
    - 串行调度策略2（S2）
  - 交错执行
    - 不可串行化的调度（S3）
    - 可串行化的调度（S4）

串行调度策略1（S1）

- 串行调度策略，正确的调度

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

串行调度策略2（S2）

- 串行调度策略，正确的调度

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

不可串行化的调度（S3）

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

- 其执行结果与S1,S2的结果都不同，所以是错误的调度。

可串行化的调度（S4）

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

- 其执行结果与串行调度S1的执行结果相同，是正确的调度。

如何保证并发操作的调度是正确的

- 为了保证并行操作的正确性，DBMS的并行控制机制必须提供一定的手段来保证调度是可串行化的。

- 从理论上讲，在某一事务执行时禁止其他事务执行的调度策略一定是可串行化的调度，这也是最简单的调度策略，但这种方法实际上是不可行的，因为它使用户不能充分共享数据库资源。

如何保证并发操作的调度是正确的

- 保证并发操作调度正确性的方法
  - 封锁方法：两段锁（Two-Phase Locking，简称2PL）协议
  - 时标方法
  - 乐观方法

两段锁协议

- 两段锁协议的内容
  - 1. 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
  - 2. 在释放一个封锁之后，事务不再获得任何其他封锁。
- “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；
  - 第二阶段是**释放封锁**，也称为收缩阶段。

两段锁协议

- 例：

T1	*T2*
Slock A	Slock A
Slock B	Unlock A
Xlock C	Slock B
Unlock B	Xlock C
Unlock A	Unlock C
Unlock C	Unlock B

- ***T1***遵守两段锁协议，而***T2***不遵守两段协议。

两段锁协议

- 并行执行的所有事务均遵守两段锁协议，则对这些事务的所有并行调度策略都是可串行化的。即，**所有遵守两段锁协议的事务，其并行执行的结果一定是正确的**
- 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
- 可串行化的调度中，不一定所有事务都必须符合两段锁协议。

两段锁协议

- 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**

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

Ssm

两段锁协议

- 两段锁协议与三级封锁协议
  - ¤ 两类不同目的的协议
    - 两段锁协议
      保证并发调度的正确性
    - 三级封锁协议
      在不同程度上保证数据一致性

  - ¤ **遵守第三级封锁协议必然遵守两段锁协议**

封锁粒度

- X锁和S锁都是加在某一个数据对象上的
- 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

封锁粒度

- 封锁对象可以很大也可以很小
  例： 对整个数据库加锁；对某个属性值加锁

- 封锁对象的大小称为封锁的**粒度**(Granularity)

- **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

封锁粒度

- 一般来说

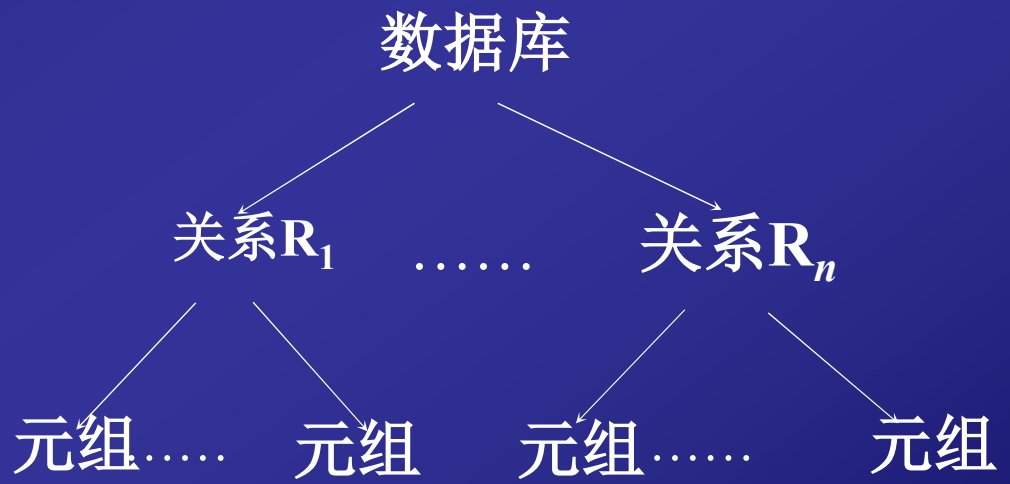
封锁的粒度越	大，	小，
系统被封锁的对象	少，	多，
并发度	小，	高，
系统开销	小，	大，

- 选择封锁粒度的原则：
  - ¤ 考虑封锁机构和并发度两个因素
  - ¤ 对系统开销与并发度进行权衡

  - ¤ 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
- 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。



Ssm

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: 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
  - 显式封锁和隐式封锁的效果是一样的

多粒度封锁协议

- 对某个数据对象加锁时系统检查的内容
  - **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。

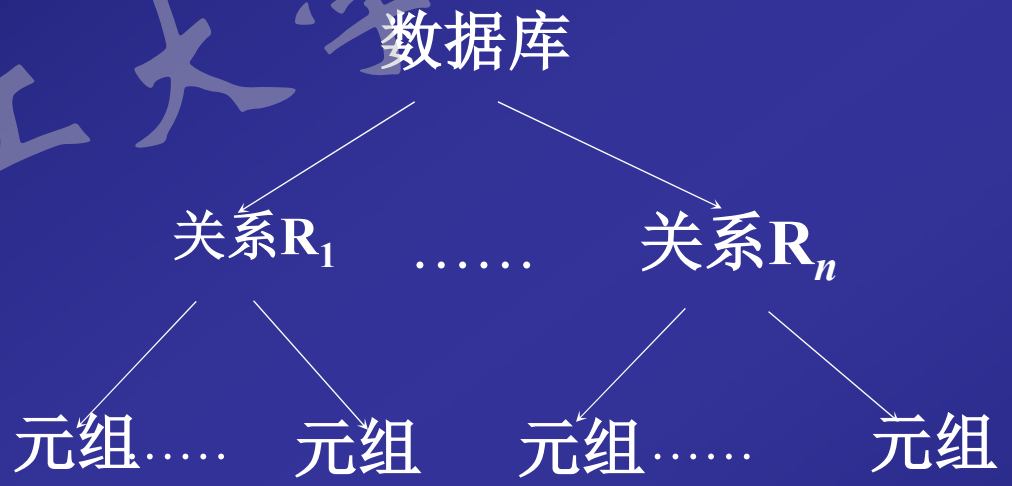
意向锁

- 意向锁（intention lock）
  - 对任一结点加基本锁，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，则说明该结点的下层结点正在被加锁

- 引进意向锁目的
  - 提高对某个数据对象加锁时系统的检查效率

意向锁

- 例：对任一元组 r 加锁，先关系R加意向锁



  - 事务T要对关系R加X锁，系统只要检查根结点数据库和关系R是否已加了不相容的锁，
  - 不需要搜索和检查R中的每一个元组是否加了X锁

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

- 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，则要首先对关系和数据库加IS锁

意向锁

- 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，则要首先对关系和数据库加IX锁。

意向锁

- 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）。

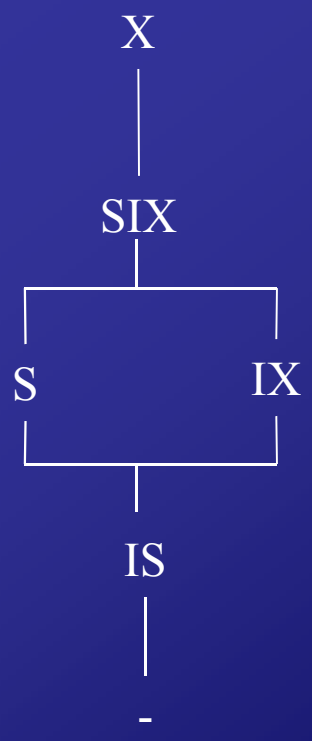
意向锁

	S	X	IS	IX	SIX
S	Y	N	Y	N	N
X	N	N	N	N	N
IS	Y	N	Y	Y	Y
IX	N	N	Y	Y	N
SIX	N	N	Y	N	N

数据锁的相容矩阵

意向锁

- 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



锁强度

意向锁

- 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T要对一个数据对象加锁，必须先对它的上层结点加意向锁

小结

- 数据共享与数据一致性是一对矛盾
  - 数据库的价值在很大程度上取决于它所能提供的数据共享度。
  - 数据共享在很大程度上取决于系统允许对数据并发操作的程度。
  - 数据并发程度又取决于数据库中的并发控制机制
  - 另一方面，数据的一致性也取决于并发控制的程度。施加的并发控制愈多，数据的一致性往往愈好。

小结

- 数据库的并发控制以事务为单位
- 数据库的并发控制通常使用封锁机制
  - 两类最常用的封锁：共享锁和排他锁
- 不同级别的封锁协议提供不同的数据一致性保证，提供不同的数据共享度。
  - 三级封锁协议

小结

- **并发控制机制调度并发事务操作是否正确的判别准则是可串行性**
  - 并发操作的正确性则通常由两段锁协议来保证。
  - 两段锁协议是可串行化调度的充分条件，但不是必要条件
- 对数据对象施加封锁，也会带来问题
  - 活锁： 先来先服务
  - 死锁：
    - 预防方法
      - 一次封锁法
      - 顺序封锁法
    - 死锁的诊断与解除
      - 超时法
      - 等待图法

小结

- 锁的粒度
  - ¤ 意向锁
- 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。但是其依据的基本原理和技术是共同的。

大连理工大学

The End

Ssm

大连理工大学