数据库系统概论

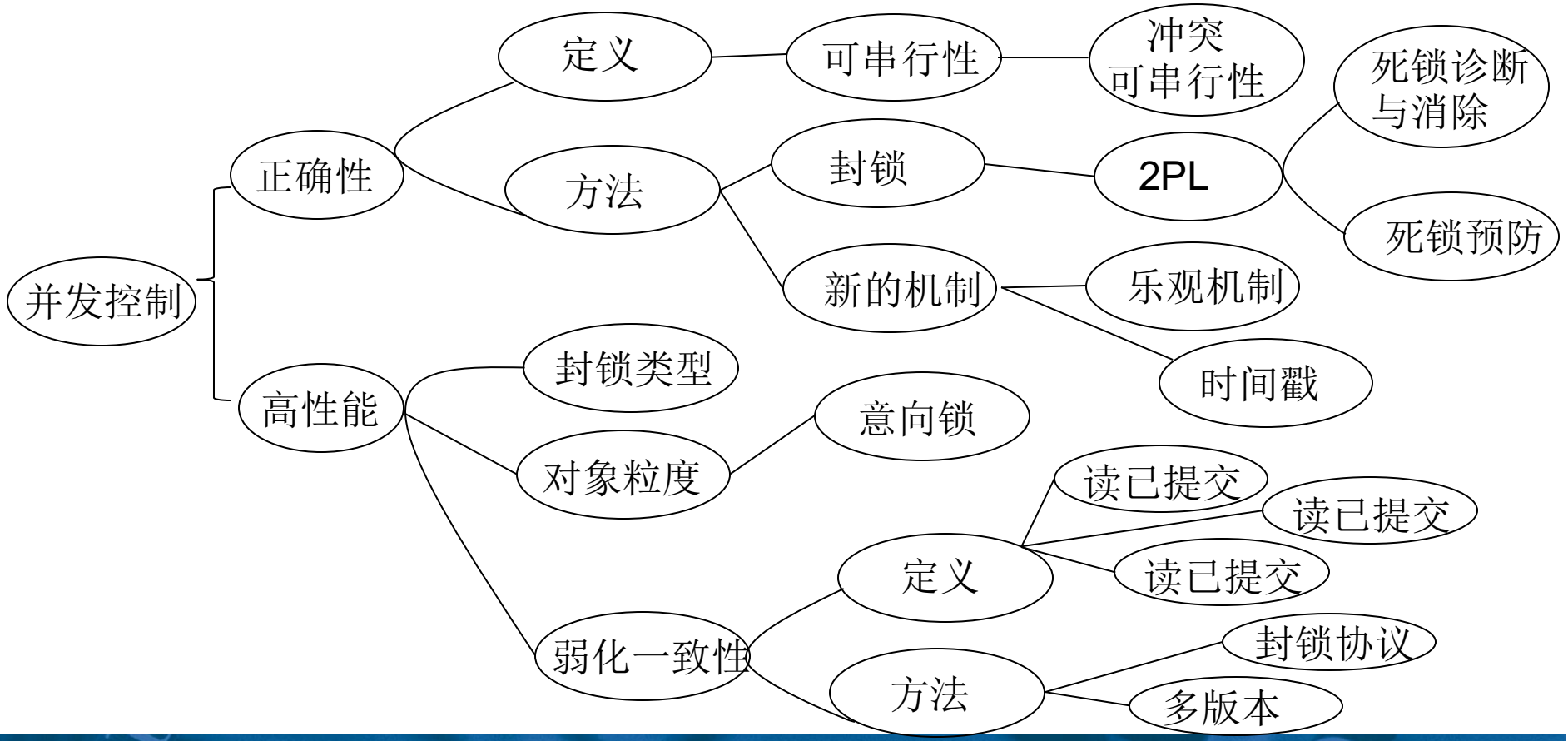
并发控制补充内容

人大信息学院：卢卫

补充内容

- 多粒度封锁
- 再议死锁预防
- 死锁诊断与消除
- 活锁

并发控制知识体系-思维导图



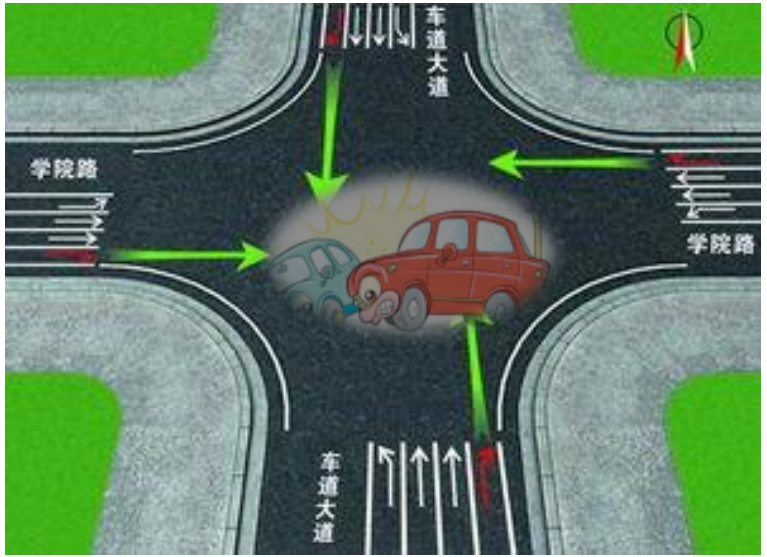
并发控制

- 并发执行的事务可能会相互干扰，不加控制会出错。
- 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）



如何提高基于封锁的方案的性能？

1. 提高并行度的方法：增加封锁的类型，使得“读读”不阻塞。
2. 引入层次封锁，减少锁表规模
3. 引入“数据一致性的分级”，弱化对一致性的要求

1 引入封锁类型

- 封锁类型
  - 排它锁（Exclusive Locks，简记为X锁）
  - 共享锁（Share Locks，简记为S锁）

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

锁的相容矩阵

T_2 \ T_1	X（20%）	S（80%）	–
X（20%）	N（0.04）	N（0.16）	Y
S（80%）	N（0.16）	Y（0.64）	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（1%）	S（99%）	–
X（1%）	N（0.0001）	N（0.0099）	Y
S（99%）	N（0.0099）	Y（0.9801）	Y
–	Y	Y	Y

Y=Yes，相容的请求
N=No，不相容的请求

2. 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

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

选择封锁粒度的原则（续）

例

- 若封锁粒度是数据页，事务T_1需要修改元组L_1，则T_1必须对包含L_1的整个数据页A加锁。如果T_1对A加锁后事务T_2要修改A中元组L_2，则T_2被迫等待，直到T_1释放A。
- 如果封锁粒度是元组，则T_1和T_2可以同时对L_1和L_2加锁，不需要互相等待，提高了系统的并行度。
- 又如，事务T需要读取整个表，若封锁粒度是元组，T必须对表中的每一个元组加锁，开销极大

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

封锁的粒度

1 多粒度封锁

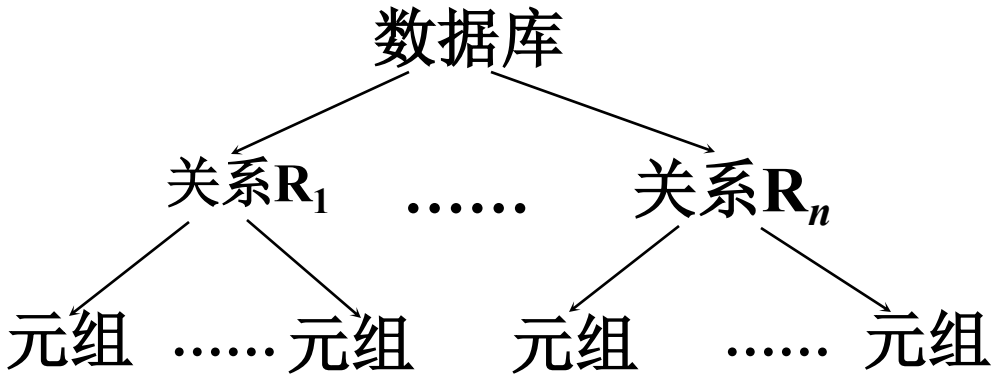
2 意向锁

1 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

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

封锁的粒度

多粒度封锁

意向锁

意向锁

- 引进意向锁（intention lock）目的
  - 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

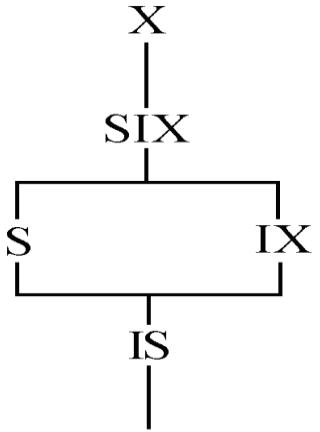
T_1 \ T_2	S	X	IS	IX	SIX	–
S	Y	N	Y	N	N	Y
X	N	N	N	N	N	Y
IS	Y	N	Y	Y	Y	Y
IX	N	N	Y	Y	N	Y
SIX	N	N	Y	N	N	Y
–	Y	Y	Y	Y	Y	Y

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

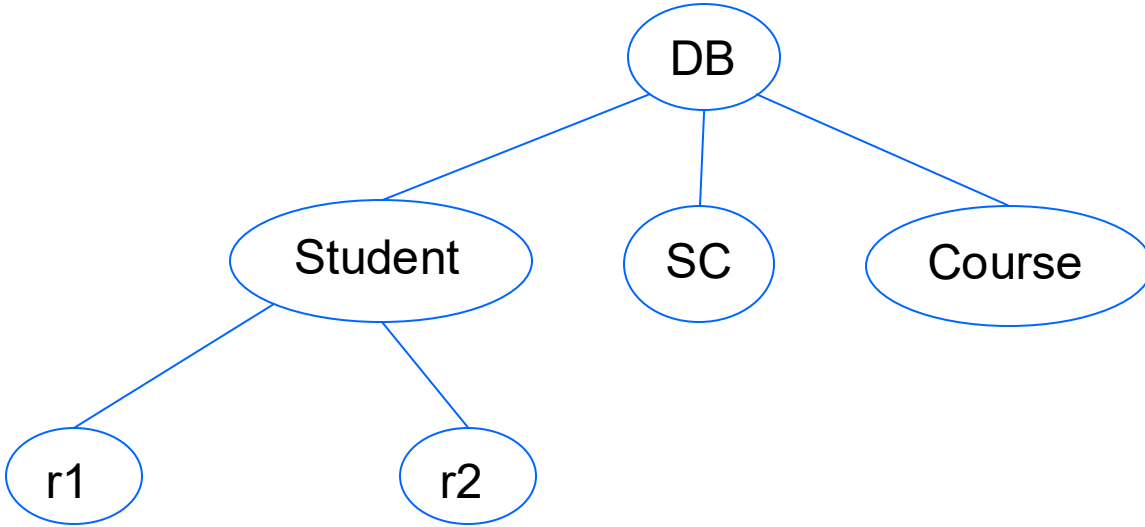
意向锁（续）

- 实例:
  - T1: select Sno, Sname from Student where Sno = ‘201701001’
  - T2: select Sno, Sname from Student
  - T3: update student set sage = sage + 1 where Sno = ‘201701002’
- 问题
  - 基于多粒度树：如何对T1事务加锁，如何对T2事务加锁；
  - 在T1加锁的基础上，如何对T3事务加锁

意向锁（续）

三级粒度树示例

r1：Student表中Sno= ‘201701001’的记录；
r2：Student表中Sno= ‘201701002’的记录；

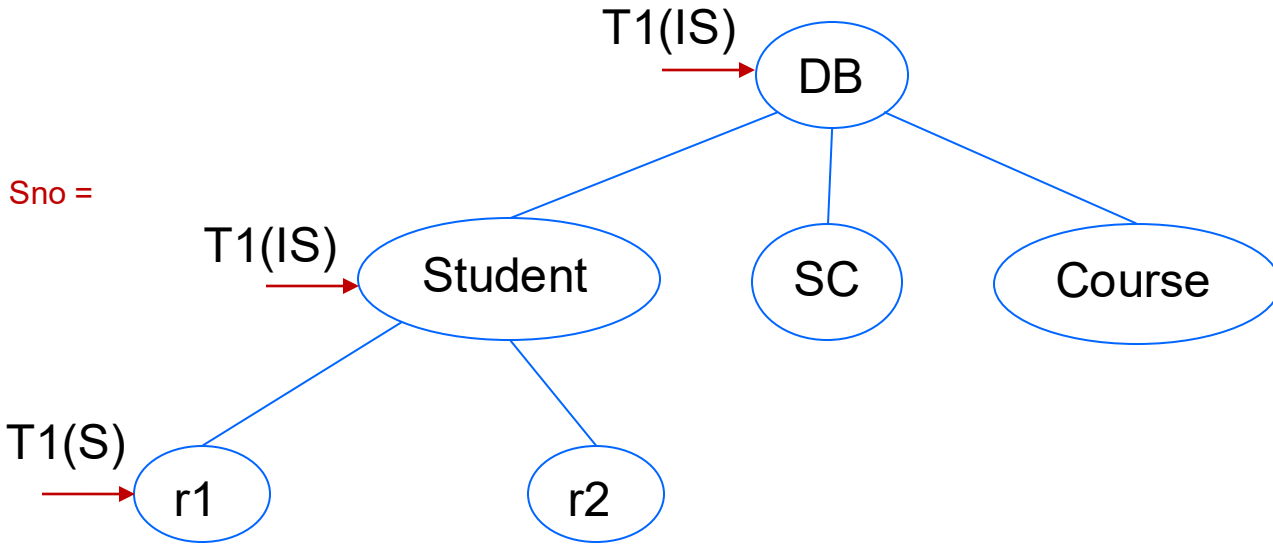


意向锁（续）

三级粒度树示例

r1：Student表中Sno= ‘201701001’的记录；
r2：Student表中Sno= ‘201701002’的记录；

T1: select Sno, Sname from Student where Sno = ‘201701001’



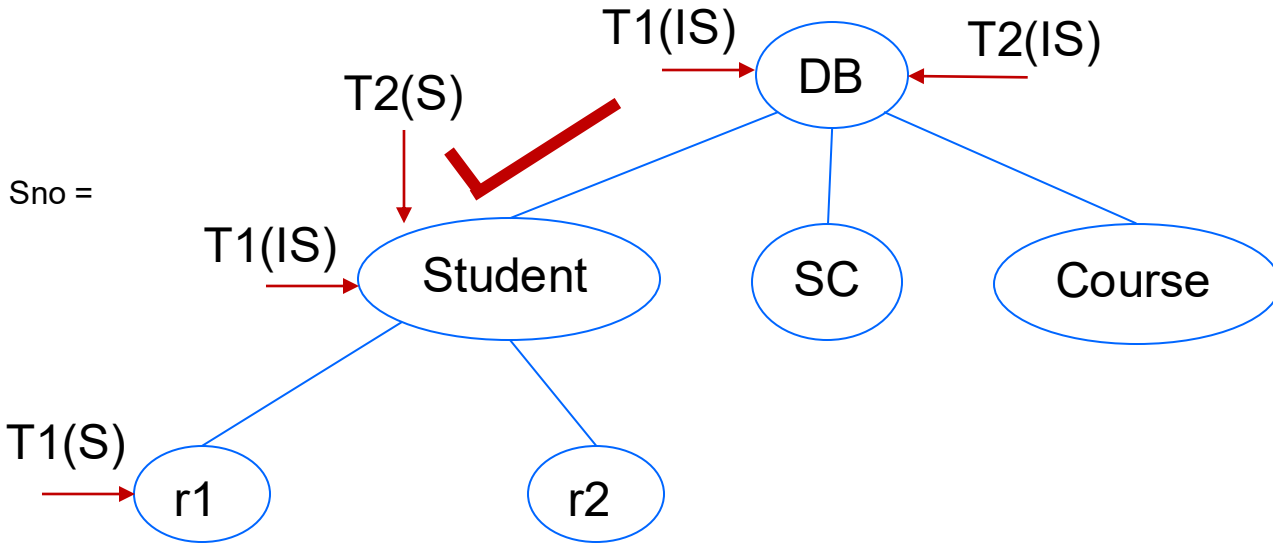
意向锁（续）

三级粒度树示例

r1：Student表中Sno= ‘201701001’的记录；
r2：Student表中Sno= ‘201701002’的记录；

T1: select Sno, Sname from Student where Sno = ‘201701001’

T2: select Sno, Sname from Student



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

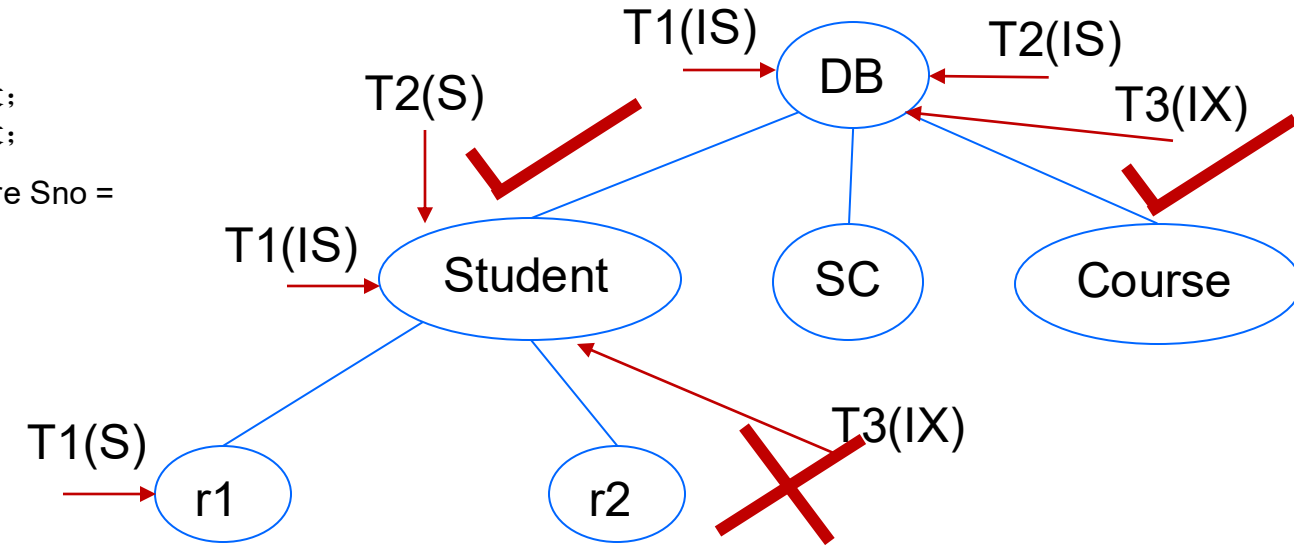
意向锁（续）

三级粒度树示例

r1：Student表中Sno= ‘201701001’的记录；
r2：Student表中Sno= ‘201701002’的记录；

T1: select Sno, Sname from Student where Sno = ‘201701001’

T2: select Sno, Sname from Student



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

T3: update student set sage = sage + 1
where Sno = ‘201701002’

重新定义“正确”的概念

- 正确=“可串行化”
- 为了效率，可否妥协？

补充内容

- 多粒度封锁
- 再议死锁预防
- 死锁诊断与消除
- 活锁

（1）一次封锁法-Calvin

- 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
- 存在的问题
  - 事务提前确定读写集
  - 批处理

T1: read(y), write(x)

T2: read(z), write(y)

T3: write(z), write(x)

Calvin 的执行实例

The locking thread performs the following:

- Lock y (SH) and x (EX) and dispatch T1 for execution
- Lock z (SH) and add T2's EX lock request into y's waiting queue
- Add T3's EX lock requests into z's and y's waiting queues

（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

（3）避免出现环

- **NO-Wait**
  - 消除边
- **Wait-Die**
  - 单向边：优先级高的等优先级低的，如果优先级低的等优先级高的，则让优先级低的回滚
- **Wound-Wait**
  - 单向边：优先级低的等优先级高的，如果优先级高的等优先级低的，则让优先级低的回滚

补充内容

- 多粒度封锁
- 再议死锁预防
- 死锁诊断与消除
- 活锁

死锁的诊断与解除

- 死锁的诊断

  （1）超时法

  （2）等待图法

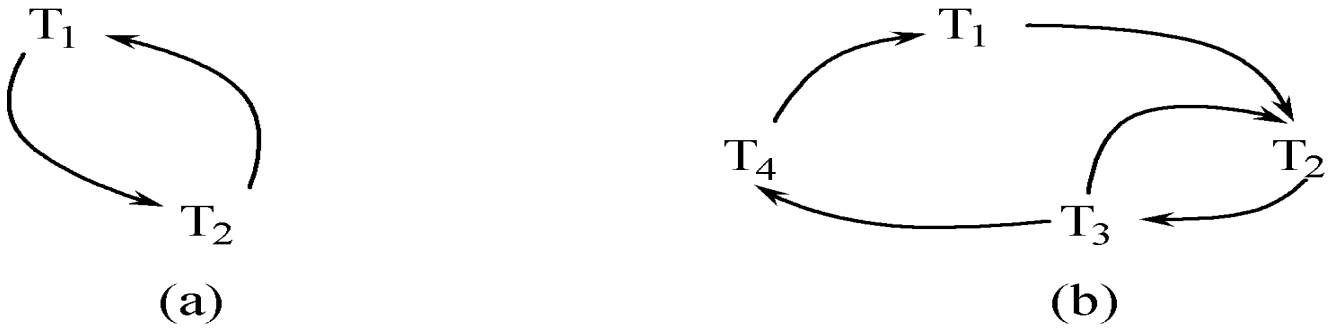
（1） 超时法

- 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。

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

补充内容

- 多粒度封锁
- 再议死锁预防
- 死锁诊断与消除
- 活锁

活锁

- 事务T_1封锁了数据R
- 事务T_2又请求封锁R，于是T_2等待。
- T_3也请求封锁R，当T_1释放了R上的封锁之后系统首先批准了T_3的请求，T_2仍然等待。
- T_4又请求封锁R，当T_3释放了R上的封锁之后系统又批准了T_4的请求……
- T_2有可能永远等待，这就是活锁的情形

活锁（续）

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

(a)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