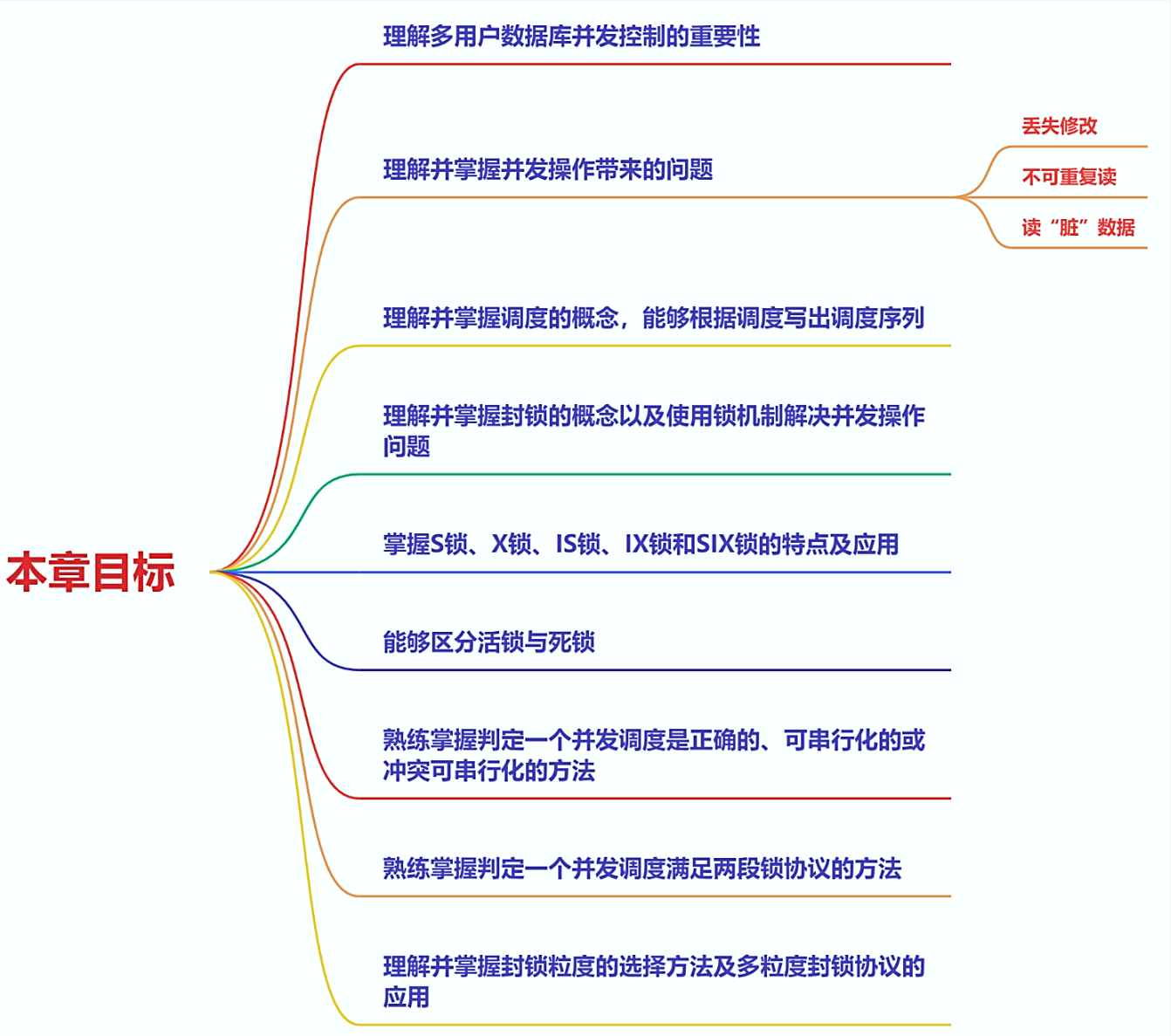
第12章 并发控制

1 本章导图


本章目标
理解多用户数据库并发控制的重要性
理解并掌握并发操作带来的问题
丢失修改
不可重复读
读“脏”数据
理解并掌握调度的概念，能够根据调度写出调度序列
理解并掌握封锁的概念以及使用锁机制解决并发操作问题
掌握S锁、X锁、IS锁、IX锁和SIX锁的特点及应用
能够区分活锁与死锁
熟练掌握判定一个并发调度是正确的、可串行化的或冲突可串行化的方法
熟练掌握判定一个并发调度满足两段锁协议的方法
理解并掌握封锁粒度的选择方法及多粒度封锁协议的应用

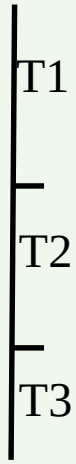
2 大纲

- 问题背景
- 并发控制概述
- 事务的隔离级别
- 封锁
- 封锁协议
- 活锁和死锁
- 并发调度的可串行性
- 两段锁协议
- 封锁的粒度
- 其他并发控制机制
- 本章小结

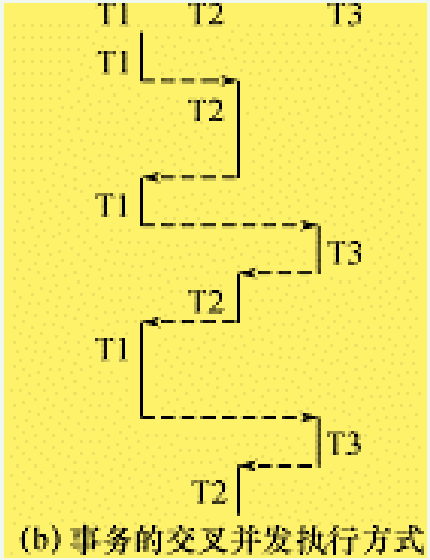
3 问题背景

- 多用户数据库系统
  - 即允许多个用户同时使用一个数据库的数据库系统
    - 飞机订票系统
    - 银行数据库系统
    - 网上购物系统
  - 特点：在同一时刻并发运行的事务数可达成千上万个

- 多事务执行方式
  1. 事务串行执行
  2. 交叉并发方式
  3. 同时并发方式



(a) 事务的串行执行方式



(b) 事务的交叉并发执行方式

同时并发方式(多处理机系统)

- 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
- 受制于硬件环境，需要更复杂的并发控制机制

3 ~(cont.)

- 当多个用户并发地存取数据库时就会产生多个事务同时存取同一数据的情况
- 若对并发操作不加控制就可能会存取和存储不正确的数据，破坏事务的一致性和数据库的一致性
- 所以，**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
- 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

4 并发控制概述

- 事务是并发控制的基本单位
  - 保证事务的**ACID**特性是事务处理的重要任务
  - 多个事务对数据库的并发操作可能破坏事务的**ACID**特性
- 为保证事务的隔离性和数据库的一致性，**DBMS**需要对并发操作进行正确的调度
  - 这是**DBMS**中并发控制机制的责任
- 调度(schedule)
  - **A list of actions**，即**reading, writing, aborting, committing**的组合序列

D=

T1	T2	T3
①R(X) ②W(X) ③COMMIT		
	④R(Y) ⑤W(Y) ⑥COMMIT	
		⑦R(Z) ⑧W(Z) ⑨COMMIT

厦門大學 XIAMEN UNIVERSITY

- 并发操作带来数据的不一致性实例

[例12.1] 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 乙售票点也卖出一张机票，修改余额A←A-1，所以A为15，把A写回数据库

- 结果明明卖出两张机票，数据库中机票余额只减少1，这种情况称为数据库的不一致性，是由并发操作引起的
- 在并发操作情况下，对T_1、T_2两个事务的操作序列的调度是随机的
- 若按上面的调度序列执行，T_1事务的修改就被丢失

- 并发操作带来的数据不一致性表现：
  - 因两个事务T_1和T_2的读-读操作不会导致数据的不一致性，故可能导致数据不一致性的T_1和T_2的数据操作分为3种情况：(T_1写,T_2写)、(T_1读,T_2写)、(T_1写,T_2读)
    - 丢失修改(lost update) --(T_1写，T_2写)情形
    - 脏读(dirty read) --(T_1写，T_2读)情形
    - 不可重复读(non-repeatable read) --(T_1读，T_2修改)情形
    - *幻读(phantom row) --(T_1读，T_2删除或插入)情形
- 记号：
  - R(x)表示事务读数据x；W(x)表示事务写数据x

4 ~(cont.)

丢失修改

- 是指两个事务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
①R(A)=16	
	②R(A)=16
③ A←A-1 W(A)=15	
	④ A←A-1 W(A)=15

示例

不可重复读

- 事务T_1读取某一数据后，事务T_2对其做了修改，当事务T_1再次读该数据时，得到与前一次不同的值；
- 事务T_1按一定条件从数据库中读取了某些数据记录后，事务T_2删除了其中部分记录，当T1再次按相同条件读取数据时，发现某些记录神秘地消失了。
- 事务T_1按一定条件从数据库中读取某些数据记录后，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**

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)	

示例

脏读

- 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。也称为读“脏”数据 。

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	

示例

4 示例1：SQL示例的脏读问题

- 脏读就是事务T_2读取了事务T_1没有提交的结果，该结果可能会回滚

T_1	T_2
BEGIN TRANSACTION; Update Student SET Sdept='IS' WHERE sno='201215123' ; --原来在MA系，现在转到IS系 ROLLBACK;	 BEGIN TRANSACTION; SELECT * FROM student WHERE sno='201215123';

4 示例2：SQL示例的不可重复读问题

- 不可重复读就是事务T_1执行更新操作，事务T_2中多个读操作返回不同的结果

T_1	T_2
BEGIN TRANSACTION; Update Student SET Sdept='IS' WHERE sno='201215123' ; --转到IS系	BEGIN TRANSACTION; SELECT * FROM student WHERE sno='201215123'; --假设为MA系 SELECT * FROM student WHERE sno='201215123';

4 示例3：SQL示例的幻影读问题

厦門大學 XIAMEN UNIVERSITY

- 幻影读与不可重复读类似，就是事务T2执行相同的SELECT后的结果不同
  - 幻影读中的T1是delete或insert操作，而不可重复读中的T1通常为update操作

T_1	T_2
	Begin transcation; Select sname from Student where sdept='IS'; --查询结果只有“张立”一条记录
Begin transaction; Insert into Student values('201215126','吴伟','男',17,'IS');	
	Select sname from Student where sdept='IS'; --查询结果出现了“张立”和“吴伟”两条记录

4 并发控制概述(cont.)

- 产生这些数据不一致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，例如有些统计工作涉及数据量很大，读到一些“脏”数据对统计精度没什么影响，可以降低对一致性的要求以减少系统开销
- 并发控制的主要技术
  - 封锁(locking)
  - 时间戳(timestamp)
  - 乐观方法（optimistic scheduler）
  - 多版本并发控制(MVCC)

5 事务的隔离级别

- 为防止数据不一致，需要DBMS对并发操作进行控制
- 这种控制越严格，事务的隔离性就越强，数据的一致性就越有保障，但系统的效率也会随之下降
- SQL标准中给出了事务的四类隔离级别，以满足不同应用场景的需求
  - 四类隔离级别由低到高分别是：**读未提交、读已提交、可重复读、可串行化**
- **读未提交(Read Uncommitted)：隔离程度最小，几乎为0，但并发度最大**
  - 允许一个事务可以读取另一个未提交事务正在修改的数据
- **读已提交(Read Committed)：很多数据库系统默认的隔离级别**
  - 只允许一个事务读其他事务已提交的数据
- **可重复读(Repeatable Read)**
  - 一个事务开始读取数据后，其他事务就不能再对该数据执行UPDATE操作了
- **可串行化(Serializable)：隔离程度最大，但并发度几乎为0**
  - 事务执行顺序是可串行化的，最高的事务隔离级别

5 ~(cont.)

表 12.1　事务隔离级别与数据不一致性的关系

事务隔离级别	数据不一致性			
	丢失修改	脏读	不可重复读	*幻读
读未提交	否	是	是	是
读已提交	否	否	是	是
可重复读	否	否	否	是
可串行化	否	否	否	否

参考：https://www.geeksforgeeks.org/transaction-isolation-levels-dbms/

验证实验(MySQL)：https://blog.csdn.net/zhangtangzhao/article/details/127481992

- 事务隔离级别并不是越高越好
  - 应根据应用的特点和需求选择合适的事务隔离级别
  - 默认事务隔离级别是“读已提交”的RDBMS产品：KingBase、SQL Server、Oracle
  - 默认事务隔离级别是“可重复读”的RDBMS产品：MySql

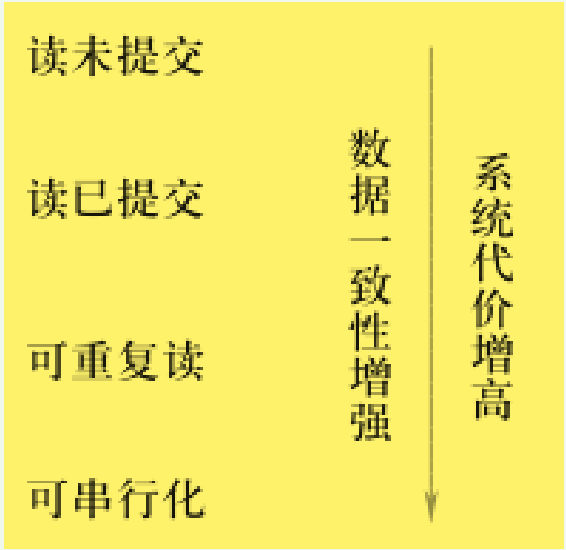


图12.3 事务隔离级别与数据一致性及系统代价的关系

封锁

- **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，其他的事务不能更新或读取此数据对象
- 封锁是实现并发控制的一个非常重要的技术
- 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**
  - 排他型锁(**exclusive locks**，简称**X**锁，写锁)
  - 共享型锁(**share locks**，简称**S**锁，读锁)

6.1排他型锁与共享型锁

排他型锁	共享型锁
• 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。	• 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。

6.2 锁的相容矩阵

- 锁的相容矩阵用于表示排他型锁和共享型锁的控制方式

T1	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

- 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
    - 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
- 三级封锁协议
  - 对并发操作的不正确调度可能会带来丢失修改、不可重复读和脏读等数据不一致性问题
  - 三级封锁协议分别在不同程度上解决了这些问题，为并发操作的正确调度提供一定的保证
  - 不同级别的封锁协议达到的数据一致性级别是不同的

7 ~(cont.)

- 3种不同级别的封锁协议：
  - **一级封锁协议**
  - **二级封锁协议**
  - **三级封锁协议**

7.1一级封锁协议

- **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脏读
  - 即，在事务已对数据加了X锁的前提下，如果其它事务不需要修改该数据，则允许其不加锁地读该数据，这点对理解后面的两个协议很重要

- 一级封锁协议解决“丢失修改”问题示例

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-1
⑤	W(A)=14
	Commit
	Unlock A

- 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。

没有丢失修改

7.2 二级封锁协议

- 二级封锁协议是指在一级封锁协议基础上，增加事务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
- 二级封锁协议可防止丢失修改和脏读
- 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

- 二级封锁协议解决脏读问题示例

T_1	T_2
① Xlock C	
R(C)=100	
C←C*2	
W(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
	R(C)=100
⑤	Commit C
	Unlock C

- 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脏读

不脏读

7.3 三级封锁协议

- 三级封锁协议是指在一级封锁协议基础上，增加事务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S锁
- 释放S锁的时机是它与二级封锁协议不同之处
- 三级封锁协议可防止丢失修改、脏读和不可重复读

- 三级封锁协议解决“不可重复读”问题示例

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

- 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T1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

可重复读

封锁协议(cont.)

- 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

表12.2 不同级别的封锁协议和一致性保证

	X锁		S锁		一致性保证			隔离性级别保证
	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“脏”读	可重复读	
一级封锁协议		√			√			读未提交
二级封锁协议		√	√		√	√		读已提交
三级封锁协议		√		√	√	√	√	可串行化

8 活锁和死锁

厦門大學 XIAMEN UNIVERSITY

- 本小节主要内容：
  - **活锁(LiveLock)**
  - **死锁(DeadLock)**

- **什么是活锁?**
  - 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

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	•	•
			•

解决方案

- 采用先来先服务的策略

什么是死锁?

- 事务T1封锁了数据R1
- T2封锁了数据R2
- T1又请求封锁R2，因T2已封锁了R2，于是T1等待T2释放R2上的锁
- 接着T2又申请封锁R1，因T1已封锁了R1，T2也只能等待T1释放R1上的锁
- 这样T1在等待T2，而T2又在等待T1，T1和T2两个事务永远不能结束，形成死锁

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
	•
	•

解决方案

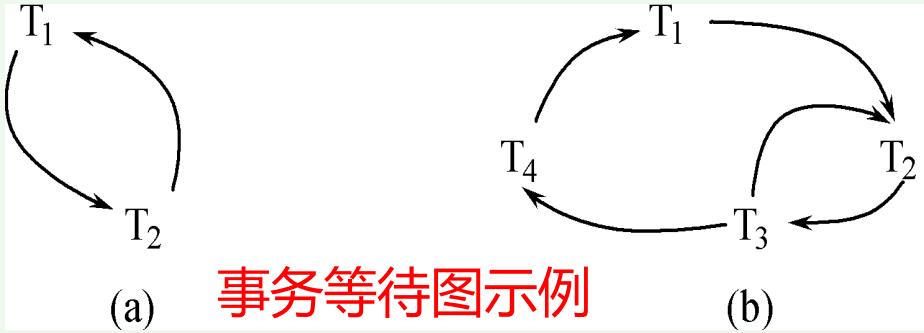
- 1.预防死锁
- 2.允许发生死锁，但定期诊断死锁；若有则解除

- 解决死锁的两类方法：
  1.死锁的预防
  2.死锁的诊断与解除
- 防止死锁的发生就是要破坏产生死锁的条件
  - 1.预防方法
    - 一次封锁法
    - 顺序封锁法

⟸ **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**

一次封锁法	顺序封锁法
• 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。 • 在死锁的例子中，如果事务T_1将数据对象R_1和R_2一次加锁，T_1就可以执行下去，而T_2等待。T_1执行完后释放R_1、R_2上的锁，T_2继续执行，这样就不会发生死锁。	• 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。 • 例如：在B树结构的索引中，可规定封锁的顺序必须从根结点开始，然后是下一级的子结点，逐级封锁。
存在的问题： • 降低了系统的并发度 • 难以事先精确确定封锁对象	**存在的问题：** • 维护封锁顺序困难，成本高 • 难以实现

- **2.死锁的诊断并解除**
  - 这是DBMS普遍采用的方法
    - 超时法
    - 等待图法



事务等待图示例

超时法	等待图法
• 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	• 用事务等待图动态反映所有事务的等待情况 • 并发控制子系统周期性地（比如每隔数秒）生成事务等待图，并进行检测。如果发现图中存在回路，则表示系统中出现了死锁
优点： • 实现简单 **缺点：** • 可能误判 • 若时限设置过长，死锁发生后不容易发现	**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，使其它事务得以继续运行下去

9 并发调度的可串行性

- DBMS对并发事务不同的调度可能会产生不同的结果
- 什么调度是正确的?
  - 串行调度是正确的
  - 可串行化调度也是正确的
- 可串行化调度
  - 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，这种调度策略称为可串行化调度
- 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，才认为是正确的调度

9 示例

[例12.2] 现在有两个事务T_1和T_2，分别包含下列操作：

事务T_1：读B；A=B+1；写回A

事务T_2：读A；B=A+1；写回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；按$T_2 \rightarrow T_1$次序执行结果为A=4，B=3

现给出对这两个事务不同的调度策略，请验证哪些调度是正确的调度

9 示例中两个并发事务的4种不同调度

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

(a)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	

(b)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

(c)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

(d)可串行化的调度

9 课堂练习

- 现在有两个事务T_1和T_2，分别包含下列操作：
  - 事务T_1：读X；X=X-N；写回X；读Y；Y=Y+N；写回Y；
  - 事务T_2：读X；X=X+M；写回X

假设X=90，Y=90，M=2，N=3。请判断以下两个调度是否正确？

T_1	T_2
R(X)	
X=X-N	
	R(X)
	X=X+M
W(X)	
R(Y)	
	W(X)
Y=Y+N	
W(Y)	

T_1	T_2
R(X)	
X=X-N	
W(X)	
	R(X)
	X=X+M
	W(X)
R(Y)	
Y=Y+N	
W(Y)	

9 冲突可串行化调度

- 如何判断一个n个并发事务的调度是正确的调度?
  - 根据正确调度的充要条件是可串行化调度，可以按如下步骤判定：
    - 计算出这n个事务的所有串行组合调度的结果，需要n！次计算；
    - 计算待判断调度的结果。若它与之前某个串行调度的结果相同，则该调度为正确的调度，否则不是正确的调度。**IF YOU ARE LUCKY , Otherwise,…**
  - 然而，上述穷举方法对于判断大量并发操作的正确调度在实际中不可行

- 冲突可串行化调度是一类充分非必要且实际可行的正确调度
  - 因此，如果能够判定一个并发调度是冲突可串行化的调度，那么该调度一定是正确的调度
  - 一个比可串行化更严格的条件，被商用系统的调度器采用

9 ~(cont.)

- 考虑一个含有分别属于T_i与T_j的两条连续指令I_i与I_j（$i \neq j$）的调度S，如果I_i与I_j引用不同的数据项，则交换I_i与I_j不会影响调度中任何指令的结果。如果I_i与I_j引用相同的数据项，在两者的顺序是重要的。四种交换情形：

  (1) I_i=R(Q), I_j=R(Q); (2) I_i=R(Q), I_j=W(Q); (3) I_i=W(Q), I_j=R(Q); (4) I_i =W(Q), I_j =W(Q)

  - 只有在I_i与I_j全为**read**指令时，两条指令的执行顺序才是无关紧要的
  - 当I_i与I_j 是不同事务对相同数据项的操作，且其中至少有一个是**write**指令时，则称是I_i与I_j冲突的

- **冲突操作**
  - 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 */
- 其他操作是不冲突操作

9 ~(cont.)

- 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
- 冲突可串行化调度
  - 一个调度SC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SC'，如果SC'是串行的，称调度SC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
  - 可用这个方法来判断一个调度是否是冲突可串行化的

[例12.3] 今有调度sc_1

$$SC_1 = r_1(A)w_1(A)\ r_2(A)w_2(A)\ \ r_1(B)w_1(B)\ r_2(B)w_2(B)$$

$$SC_2 = \underbrace{r_1(A)w_1(A)\ r_1(B)w_1(B)}_{T_1}\ \underbrace{r_2(A)w_2(A)r_2(B)w_2(B)}_{T_2}$$

SC_2等价于一个串行调度T_1，T_2，所以SC_1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而非必要条件

- 存在不满足冲突可串行化条件的可串行化调度。

  [例12.4] 有三个事务$T_1=W_1(Y)W_1(X)$，$T_2=W_2(Y)W_2(X)$，$T_3=W_3(X)$。

  – 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（为什么？）。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

[例] 判断以下两个调度是否为冲突可串行化调度？（之前练习的两个调度）

T_1	T_2
R(X)	
X=X-N	
	R(X)
	X=X+M
W(X)	
R(Y)	
	W(X)
Y=Y+N	
W(Y)	

T_1	T_2
R(X)	
X=X-N	
W(X)	
	R(X)
	X=X+M
	W(X)
R(Y)	
Y=Y+N	
W(Y)	

[解]：首先将两个调度分别改写成：

- $R_1(X)R_2(X)W_1(X)R_1(Y)W_2(X)W_1(Y)$ ⇒ 不是冲突可串行化调度
- $R_1(X)\ W_1(X)R_2(X)W_2(X)R_1(Y)W_1(Y)$ ⇒ 是冲突可串行化调度

- **优先图算法**—冲突可串行化调度的判定方法

1. 构造一个有向图G=(N, E)，其中
   - N={ $T_1, T_2, \ldots, T_n$ }，T_i（i = 1, 2, …, n）表示事务T_i对应的结点；
   - E为不同结点之间有向边的集合
   - 有向边e_{ij}：$T_i \rightarrow T_j$的定义如下：如果出现以下任一种情形，则结点T_i到结点T_j（i≠j）之间存在一条有向边
     - $W_i(X)R_j(X)$，$R_i(X)W_j(X)$，$W_i(X)W_j(X)$
2. 该调度为冲突可串行化调度的充分必要条件为有向图G不存在回路

[例] 利用优先图算法判定以下调度是否为冲突可串行化调度?

- SC_1: $R_1(X)R_2(X)W_1(X)R_1(Y)W_2(X)W_1(Y)$ ---前例的两个调度
- SC_2: $R_1(X)\ W_1(X)R_2(X)W_2(X)R_1(Y)W_1(Y)$ ---现在使用优先图算法来验证结论

- **课堂练习:**

画出以下的调度的优先图，并判断是否为冲突可串行化调度?

① $r_1(X)r_3(X)w_1(X)r_2(X)w_3(X)$

② $r_3(X)r_2(X)w_3(X)r_1(X)w_1(X)$

10 两段锁协议

- **DBMS普遍采用两段锁协议(two-phase lock, 2PL)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
    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，事务不再申请和获得任何其他封锁
- 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(growing phase)
    - 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(shrinking phase)
    - 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

例如，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　　收缩阶段

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

事务T_j不遵守两段锁协议的封锁序列

结论：

- 若并发执行的所有事务都遵守两段锁协议，则对这些事务的任何并发调度都是可串行化的

10 ~(cont.)

事务T_1	事务T_2
SLOCK A	
R(A)=260	
	SLOCK C
	R(C)=300
XLOCK A	
W(A)=160	
	XLOCK C
	W(C)=250
	SLOCK A
SLOCK B	等待
R(B)=1000	等待
XLOCK B	等待
W(B)=1100	等待
UNLOCK A	等待
	R(A)=160
	XLOCK
UNLOCK B	
	W(A)=210
	UNLOCK C
	UNLOCK A

- 事务T_1的封锁序列：

Slock A　Xlock A　Slock B　Xlock B　Unlock A　Unlock B

- 事务T_2的封锁序列：

Slock C　Xlock C　Slock A　Xlock A　Unlock C

调度L_1

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

调度L_2

$L_2=R_1(A)W_1(A)R_1(B)W_1(B)R_2(C)W_2(C)R_2(A)W_2(A)$

可串行化调度

10 ~(cont.)

- 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(图12.8d)
- 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

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

发生死锁

10 可串行化调度小结

- 正确的调度 ⇔ 可串行化调度
  - 正确的调度 ⇔ 串行调度结论正确吗?
- 三类可串行化调度的充分非必要条件
  - 串行调度
  - 冲突可串行化调度
  - 遵守两段锁协议的调度
- 判定冲突可串行化调度的方法
  - 优先图算法
  - 利用该算法可以很容易地构造冲突可串行化调度的例子
- 遵守两段锁协议可能发生死锁

openGauss的并发控制机制

- 官网：
  - MOT乐观并发控制

https://www.opengauss.org/zh/docs/3.0.0/docs/Developerguide/MOT%E4%B9%90%E8%A7%82%E5%B9%B6%E5%8F%91%E6%8E%A7%E5%88%B6.html

  - 锁

https://www.opengauss.org/zh/docs/3.0.0/docs/BriefTutorial/%E9%94%81.html

11 封锁的粒度

- 封锁对象的大小称为封锁粒度(lock granularity)
  - 粗粒度(coarse granularity) VS. 细粒度(fine granularity)
- 封锁的对象
  - 逻辑单元
    - RDBMS的属性值、属性值的集合、元组、关系、索引项、整个索引、整个数据库
  - 物理单元
    - RDBMS的页(数据页或索引页)、物理记录
- 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，并发度越高，但系统开销也就越大

[示例]:

- 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
- 如果事务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

- 封锁粒度的选择
  - 选择封锁粒度时应同时考虑封锁开销和并发度两个因素，适当选择封锁粒度以求得最优效果

一般考虑:

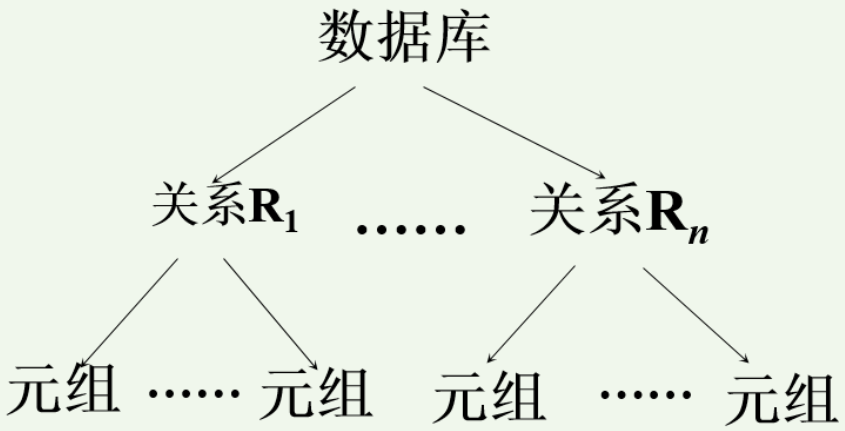
- 需要处理某个关系的大量元组的用户事务 — 以关系为封锁粒度
- 需要处理多个关系的大量元组的用户事务 — 以数据库为封锁粒度
- 只处理少量元组的用户事务 — 以元组为封锁粒度

- **多粒度封锁**
  - 在一个系统中同时支持多种封锁粒度供不同的事务选择的封锁方法称为多粒度封锁
- 原因：
  - 在封锁机制中，某些情况需要把多个数据项聚为一组，将它们作为一个同步单元，这样效果可能更好
    - 例如，如果事务T_i需要访问整个数据库，而且使用一种封锁协议，则事务T_i必须给数据库中每个数据项加锁。显然，执行这些加锁操作是很费时的。要是T_i能够只发出一个封锁整个数据库的加锁请求，那会更好
  - 另一方面，如果事务T_j只需存取少量数据项，就不应要求给整个数据库加锁，否则并发性就丧失了
  - 所以需要允许定义多级粒度的机制，通过允许各种大小的数据项并定义数据粒度的层次结构，其中小粒度数据项嵌套在大粒度数据项中，这就是多粒度封锁机制

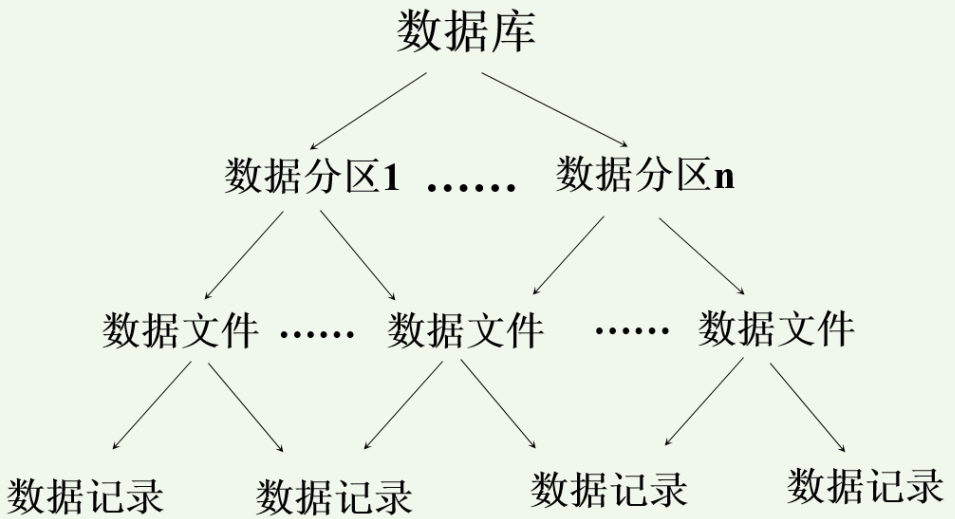
11 封锁的粒度(cont.)

- 本小节主要内容：
  - **多粒度封锁**
  - **意向锁**

- **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



三级粒度树



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
  - 指应事务的要求直接加到数据对象上的封锁
- 隐式封锁
  - 指该数据对象没有被独立加锁，是由于其上级结点加锁而使该数据对象加上了锁，即继承了来自上级结点的相同类型的锁
- 显式封锁和隐式封锁的效果是一样的

11.2意向锁

- 系统检查封锁冲突时不仅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以及R_1的下级结点，即R_1中的每一个元组，上下搜索。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

- 一般地，对某个数据对象加锁，系统要检查该数据对象上有无显式封锁与之冲突；再检查其所有上级结点，看本事务的显式封锁是否与该数据对象上的隐式封锁冲突（由于上级结点已加的封锁造成的）；看它们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

- 这种检查效率很低，为此引进意向锁(intention lock) 以提高系统的检查效率

- 意向锁的含义是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
  - 对任一结点加锁时，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
  - 有了意向锁，DBMS就无须逐个检查下一级结点的显式封锁
- 三种常用的意向锁
  - 意向共享型锁(intent shared lock，简称IS锁)
  - 意向排它型锁(intent exclusive lock，简称IX锁)
  - 共享意向排它型锁(shared and intention exclusive lock，简称SIX锁

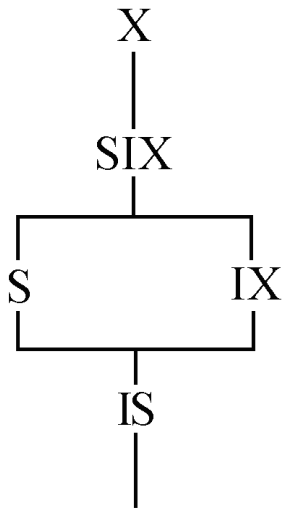
意向锁类型	特点
IS锁	• 如果对一个数据对象加IS锁，表示它的后裔结点拟（意向）加S锁 • 例如：事务T_1要对R_1中某个元组加S锁，则要首先对关系R_1和数据库加IS锁
IX锁	• 如果对一个数据对象加IX锁，表示它的后裔结点拟（意向）加X锁 • 例如：事务T_1要对R_1中某个元组加X锁，则要首先对关系R_1和数据库加IX锁
SIX锁	• 如果对一个数据对象加SIX锁，表示对它加S锁，再加IX锁，即SIX = S+IX • 例如：对某个表加SIX锁，则表示该事务要读整个表(所以要对该表加S锁)，同时会更新个别元组（所以要对该表加IX锁）

- 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

注：Y=Yes，表示相容的请求；N=No，表示不相容的请求

(a) 封锁类型的相容矩阵



(b) 锁的强度的偏序关系

- **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

- **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
    - 检查数据库是否已加了不相容的锁(X锁)
    - 然后检查R_1是否已加了不相容的锁(X或IX锁)
    - **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)
- 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DBMS产品中得到广泛应用

[例] 考虑下面的三级粒度树，根结点是整个数据库D，包括关系R_1、R_2、R_3，分别包括元组r_1，r_2，…，r_{100}，r_{101}，…，r_{200}和r_{201}，…，r_{300}，使用具有意向锁的多粒度封锁方法，对于下面的操作说明产生加锁请求的锁类型和顺序.

① 读元组r_{50}

② 读元组r_{90}到r_{210}

③ 读R_2的所有元组并修改满足条件的元组

④ 删除所有元组

[解]:

① D上加IS锁；R_1上加IS锁；r_{50}上加S锁

② D上加IS锁；R_1上加IS锁；R_2上加S锁；R_3上加IS锁；r_{90}到r_{100}上加S锁，r_{201}到r_{210}上加S锁

③ D上加IS锁和IX锁；R_2上加SIX锁

④ D上加IX锁；R_1、R_2、R_3上加X锁

厦門大學 XIAMEN UNIVERSITY

- 问答题：

1. 意向锁中为什么存在SIX锁，而没有XIS锁？

2. 完整性约束是否能够保证数据库中处理多个事务时处于一致状态？

1.如果对数据对象加SIX锁，表示对它加S锁，再加IX锁，即对数据对象加S锁，后裔结点拟加X锁。X锁与任何其他类型的锁都不相容，如果数据对象被加上X锁，后裔结点不可能被以任何锁的形式访问，因此XIS锁没有意义。
2.完整性约束能够保证操作后的数据满足某种约束条件，并不能使多个事务被正确调度，无法保证数据库处于一致性状态

12 其他并发控制机制

厦門大學 XIAMEN UNIVERSITY

- 其他并发控制方法
  - **时间戳方法**
  - **乐观方法**
  - **多版本并发控制（MVCC）**

并发控制方法	特点
时间戳方法	• 如果给每个事务盖上一个时标，即事务的开始时间 • 每个事务具有唯一的时间戳，并按照这个时间戳解决事务的冲突 • 如果发生冲突操作，就回滚具有较早时间戳的事务 • 被回滚的事务被赋予新的时间戳并从头开始执行
乐观方法	• 又称为验证方法（certifier） • 认为事务执行时很少发生冲突，因此不对事务进行特殊的管制 • 而是让它自由地执行，事务提交前再进行正确性检查 • 如果检查后发现该事务执行中出现过冲突并影响了可串行性，则拒绝提交并回滚该事务
多版本并发控制	• MVCC（multi-version concurrency control）是指在数据库中通过维护数据对象的多个版本信息，来实现高效并发控制的一种策略

12.1多版本并发控制方法

- **版本**
  - 是指数据库中数据对象的一个快照，记录了数据对象，某个时刻的状态
  - 可以为数据库系统的数据对象保留多个版本，以提高系统的并发操作程度

[例] 有一个数据对象A=5、一个写事务T_1(先启动)和一个读事务T_2(后启动)

按照传统的封锁协议：

- T_2事务必须等待事务T_1执行结束释放A上的封锁后才能获得对A的封锁
- 即，T_1和T_2实际上是串行执行的

事务T_1	事务T_2
XLOCK A	
R (A)=5	
W(A)=6	
	SLOCK A
	等待
COMMIT	等待
UNLOCK A	等待
	SLOCK A
	R (A)=6
	COMMIT
	UNLOCK A

(a) 封锁方法

事务T_1	事务T_2
R(A)=5	BEGIN TRANSACTION
W(A)=6	
创建新版本A'=6	
COMMIT	
	R(A')=6
	COMMIT

(b) MVCC

多版本机制：

- 在T_1准备写A的时候，为A生成一个新的版本(表示为A')，T_2事务不是等待，而可以继续在A'上执行
- T_2准备提交时，看一下事务T_1是否已经完成
- 如果T_1已经完成了，T_2就可以放心地提交
- 如果T_1还没有完成，那么T_2必须等待直到T_1完成

- 在多版本并发控制方法中
  - **write(Q)**操作：创建**Q**的一个新版本
  - 一个数据对象有一个版本序列Q_1，Q_2，…，Q_m,与之相关联
  - 每一个版本Q_k拥有版本的值、创建Q_k的事务的时间戳**W-timestamp(Q_k)**，和成功读取Q_k的事务的最大时间戳**R-timestamp(Q_k)**
    - **W-timestamp(Q)**表示在数据项**Q**上成功执行**write(Q)**操作的所有事务中的最大时间戳，**R-timestamp(Q)**表示在数据项**Q**上成功执行**read(Q)**操作的所有事务中的最大时间戳
  - **TS(T)**表示事务**T**的时间戳，**TS(T_i)<TS(T_j)**表示事务T_i在事务T_j之前开始执行
- 多版本协议描述（假设版本Q_k具有小于或等于**TS(T)**的最大时间戳）：
  - 若事务**T**发出**read(Q)**，则返回版本Q_k的内容
  - 若事务**T**发出**write(Q)**，则：
    - 当**TS(T) < R-timestamp(Q_k)**时，回滚**T**；
    - 当**TS(T) = W-timestamp(Q_k)**时，覆盖Q_k的内容
  - 否则，创建**Q**的新版本
  - 若一个数据对象的两个版本Q_k和Q_l，其**W-timestamp**都小于系统中最老的事务的时间戳，那么这两个版本中较旧的那个版本将不再被用到，可以从系统中删除

- 多版本并发控制利用物理存储上的多版本来维护数据的一致性
- 多版本并发控制和封锁机制相比，主要的好处是消除了数据库中数据对象读和写操作的冲突，有效地提高了系统的性能
- 多版本并发控制方法有利于提高事务的并发度，但也会产生大量的无效版本，而且在事务的结束时刻，其所影响的元组的有效性不能马上确定

12.2改进的多版本并发控制方法

- 区分事务的类型：
  - 只读事务：发生冲突的可能性很小，可以采用多版本时间戳
  - 更新事务：采用较保守的两阶段封锁(2PL)协议
- 以上两种混合协议称为MV2PL
- MV2PL的具体做法：
  - 引进一个新的封锁类型：验证锁(certify-lock,或C锁)
  - 封锁的相容矩阵如图

T_1	T_2		
	S	X	C
S	Y	Y	N
X	Y	N	N
C	N	N	N

注：Y=Yes，表示相容的请求；N=No，表示不相容的请求

图 12.14　验证锁的相容矩阵

- 读锁和写锁变得是相容的了
- 当某个事务写数据对象的时候，允许其他事务读数据（写操作将生成一个新的版本，而读操作就是在旧的版本上读）
- 一旦写事务要提交的时候，必须首先获得在那些加了写锁的数据对象上的验证锁
- 为了得到验证锁，写事务需要延迟它的提交，直到所有被它加上写锁的数据对象都被所有那些正在读它们的事务释放
- 一旦写事务获得验证锁，系统丢弃数据对象的旧值，代之以新版本，然后释放验证锁，提交事务
- 系统最多只要维护数据对象的两个版本，多个读操作可以和一个写操作并发执行，提高了读写事务之间的并发度
- MV2PL把封锁机制和时间戳方法相结合，维护一个数据的多个版本
- MV2PL和封锁机制相比，多版本并发控制对读锁与写锁要求不冲突，所以读不会阻塞写，写也不阻塞读，从而有效地提高了系统的并发性

13 本章小结

- DBMS必须提供并发控制机制来协调并发用户的并发操作以保证并发事务的隔离性和一致性，保证数据库的一致性
- 数据库的并发控制以事务为单位，通常使用封锁技术实现并发控制
- 三级封锁协议解决丢失修改、不可重复读、脏读三类问题
- 活锁与死锁的产生与解决
- 并发调度的正确与可串行性
  - 串行调度、可串行化调度、冲突可串行化调度、两段锁协议
    - 串行调度、冲突可串行化调度与两段锁协议都是可串行化调度的充分非必要条件
- 封锁粒度的选择
- 多粒度封锁与多粒度封锁协议
- 其他并发控制机制
  - 时间戳方法
  - 乐观方法
  - 多版本并发控制

14 本章小结

- 教材第十二章全部习题.
- 要求：作业布置后一周内完成并提交到课程网站