第11章 并发控制

本章目标

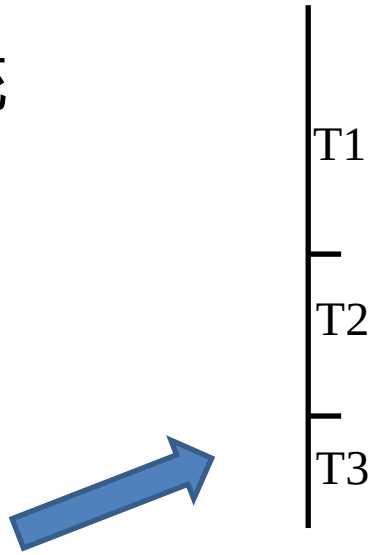
- 完成本章的学习，你应该能够
  - 理解多用户数据库并发控制的重要性
  - 理解并掌握丢失修改、不可重复读、读“脏”数据的含义及三级封锁协议解决原理
  - 理解并掌握调度的概念并能够根据并发调度写出调度序列
  - 理解并掌握封锁的概念
  - 掌握S锁、X锁、SIX锁、IX锁、IS锁的特点及应用
  - 掌握活锁和死锁的概念、特点及解决途径
  - 知道如何判断一个并发调度是正确的、可串行化的、冲突可串行化
  - 知道如何判断一个并发调度满足两段锁协议
  - 理解并掌握封锁粒度的选择及多粒度封锁协议的应用

大纲

- 背景
- 并发控制概述
- 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
- 本章小结

背景

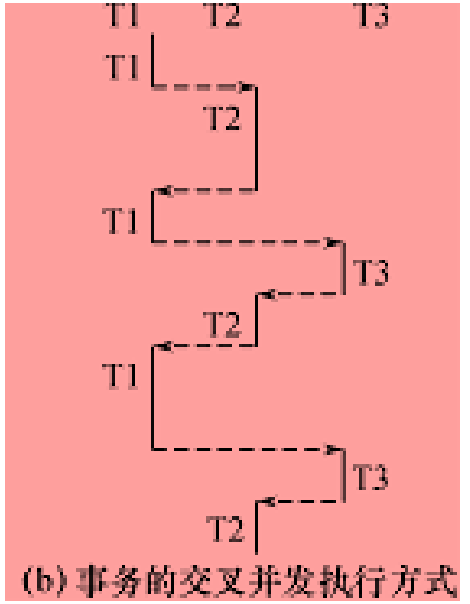
- 多用户数据库系统
  - 即允许多个用户同时使用一个数据库的数据库系统
  - 例：飞机订票数据库系统、银行数据库系统
  - 特点：
    - 在同一时刻并发运行的事务数可达数百上千个
- 事务的串行执行
  - 即每个时刻只有一个事务运行，其他事务必须等到这个事务结束以后方能运行



事务的串行执行方式

- 如果事务串行执行，则许多系统资源将处于空闲状态，因此为了充分利用系统资源，发挥数据库共享资源的特点，应该允许多个事务并行执行。

- 事务的并行执行
  - 交叉并发方式(单处理机系统)
    - 并行事务的并行操作轮流交叉运行
    - 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



(b)事务的交叉并发执行方式

  - 同时并发方式(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
    - 最理想的并发方式，但受制于硬件环境

- 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。

- 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。

大纲

- 背景
- **并发控制概述**
- 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
- 本章小结

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。
- DBMS并发控制的责任
  - 保证事务的隔离性
  - 保证事务的一致性
  - 对并发操作进行正确的调度

- 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

- 并发操作带来数据不一致性

[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T1和T2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T1和T2的数据操作分为3种情况：(T1写,T2写)、(T1读,T2写)、(T1写,T2读)

    ① 丢失修改(lost update) // (T1写，T2写)情形

    ② 不可重复读(non-repeatable read) // (T1读，T2写)情形

    ③ 读“脏”数据(dirty read) // (T1写，T2读)情形

- 记R(x)表示事务读数据x；W(x)表示事务写数据x

丢失修改

- 是指两个事务T1和T2读入同一数据并修改，T2的提交结果破坏了T1提交的结果，导致T1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-1 W(A)=15

示例

不可重复读

- 是指事务T1读取数据后，事务T2 执行更新操作，使T1无法再现前一次读取结果。

不可重复读包括三种情况：

- 事务T1读取某一数据后，事务T2对其做了修改，当事务T1再次读该数据时，得到与前一次不同的值；
- 事务T1按一定条件从数据库中读取了某些数据记录后，事务T2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T1按一定条件从数据库中读取某些数据记录后，事务T2插入了一些记录，当T1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

读”脏”数据

- 是指事务T1修改某一数据，并将其写回磁盘，事务T2读取同一数据后，T1由于某种原因被撤销，这时T1已修改过的数据恢复原值，T2读到的数据就与数据库中的数据不一致，T2读到的数据就为“脏”数据，即不正确的数据。

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

- 不可重复读示例：

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)	

- 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

并发控制的主要技术

- 并发控制机制就是利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。
- 主要技术：
  - 封锁(Locking)
  - 时间戳(Timestamp)
  - 乐观控制法
  - 多版本并发控制(MVCC)

大纲

- 背景
- 并发控制概述
- **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
- 本章小结

封锁

- 封锁是实现并发控制的一个非常重要的技术。
- 所谓封锁就是事务T在对某个数据对象(如表、记录)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。
- 确切的控制由封锁的类型决定。
- 基本的封锁类型
  - 排他锁(exclusive locks，简称X锁)
  - 共享锁(share locks，简称S锁)

- 排他锁(写锁)与共享锁(读锁)

排他锁	共享锁
• 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。

- 锁的相容矩阵(compatibility matrix)
  - 用于表示排他锁和共享锁的控制方式

T1 \ T2	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

大纲

- 背景
- 并发控制概述
- 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
- 本章小结

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。
    - 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。
- 本节介绍三级封锁协议

1.一级封锁协议

- 一级封锁协议是指事务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。
  - 事务结束包括正常结束(COMMIT)和非正常结束(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不读“脏”数据。

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

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读“脏”数据。
- 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“读脏数据”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T1在对C进行修改之前，先对C加X锁，修改其值后写回磁盘
- T2请求在C上加S锁，因T1已在C上加了X锁，T2只能等待
- T1因某种原因被撤销，C恢复为原值100
- T1释放C上的X锁后T2获得C上的S锁，读C=100。避免了T2读“脏”数据

不读“脏”数据

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、读“脏”数据和不可重复读。

二级封锁协议解决“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T1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T2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
- T1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
- T1结束才释放A，B上的S锁。T2才获得对B的X锁

可重复读

- 三类封锁协议的主要区别
  - 是在于什么操作需要申请封锁
  - 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11.1 不同级别的封锁协议和一致性保证

	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		√		√	√	√	√

大纲

- 背景
- 并发控制概述
- 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
- 本章小结

活锁和死锁

- 活锁

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	•	•
			•

- 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

解决方案：

- 采用先来先服务的策略

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
	•
	•

- 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

解决方案：

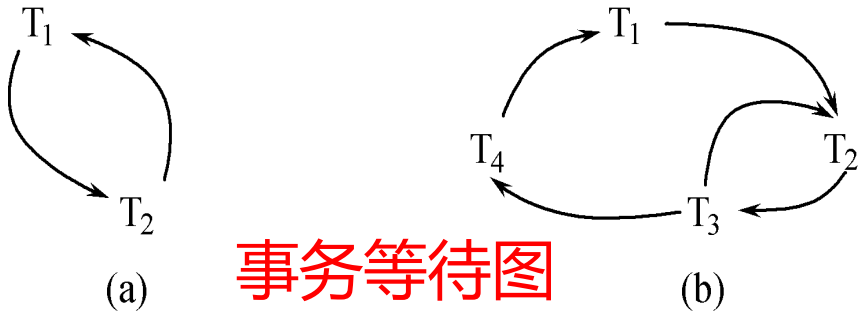
- 预防死锁
- 允许发生死锁，但定期诊断死锁；若有则解除

- 死锁的预防
  - 防止死锁的发生就是要破坏产生死锁的条件。
  - 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。 • 在活锁的例子中，如果事务T1将数据对象R1和R2一次加锁， T1就可以执行下去，而T2等待。T1执行完后释放R1、R2上的锁，T2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。 • 例子：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 • 难于事先精确确定封锁对象	**存在的问题：** • 维护成本高 • 难于实现

- 诊断并解除死锁是DBMS普遍采用的方法
  - 超时法
  - 等待图法



事务等待图

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

大纲

- 背景
- 并发控制概述
- 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
- 本章小结

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。
- 什么调度是正确的?
  - 串行调度
  - 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，称这种调度策略为可串行化(serializable)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，才认为是正确调度

[例11.2] 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

假设A、B的初值均为2。按T1→T2次序执行结果为A=3，B=4；按T2→T1次序执行结果为A=4，B=3

- 问题：如何判断一个n个并发事务的调度是正确的调度?

1. 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；
2. 计算待判断调度的结果。若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。

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

图：两个并发事务的4种不同调度

课堂练习

- 现在有两个事务，分别包含下列操作：
  - 事务T1：读X；X=X-N；写回X；读Y；Y=Y+N；写回Y；
  - 事务T2：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?

T1	T2
R(X)	
X=X-N	
	R(X)
	X=X+M
W(X)	
R(Y)	
	W(X)
Y=Y+N	
W(Y)	

T1	T2
R(X)	
X=X-N	
W(X)	
	R(X)
	X=X+M
	W(X)
R(Y)	
Y=Y+N	
W(Y)	

正确调度的判定

- 正确的调度⇔可串行化调度
  - 问题：如何判断调度是可串行化的?
    - 是穷举法对于大量并发调度存在的情况下判断其是否正确在实际中不可行
    - 冲突可串行化调度是一类重要的、实际可行的、充分非必要的正确调度
- 冲突可串行化调度
  - 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)

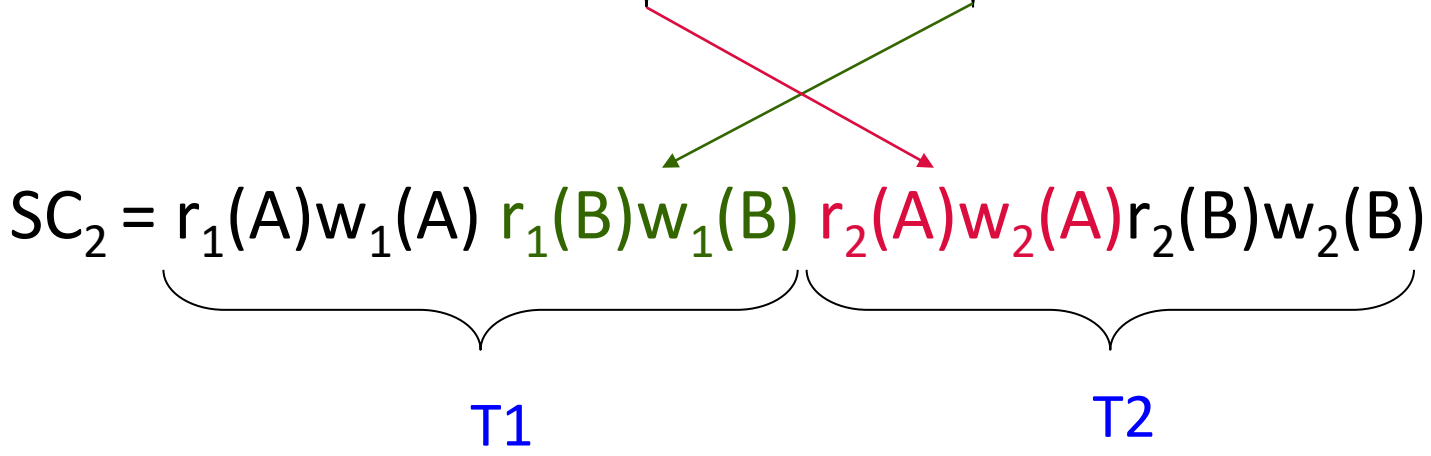
- 只有在I_i与I_j全为read指令时，两条指令的执行顺序才是无关紧要的
- 当I_i与I_j是不同事务对相同数据项的操作，且其中至少有一个是write指令时，则称是I_i与I_j冲突(conflict)的.
- 冲突操作
  - 指不同的事务对同一数据的读写操作和写写操作
    - Ri(x)与Wj(x)　/* 事务Ti读x，Tj写x，其中i≠j */
    - Wi(x)与Wj(x)　/* 事务Ti写x，Tj写x，其中i≠j */
- 其他操作是不冲突操作
  - 事务对不同数据的操作；
  - 同一事务对同一数据的读读操作

- 不同事务的冲突操作和同一事务的两个操作是不能交换(swap)的。

- 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11.3] 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等价于一个串行调度T1，T2，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，不是必要条件。
- 存在不满足冲突可串行化条件的可串行化调度。

[例11.4] 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?

T1	T2
R(X)	
X=X-N	
	R(X)
	X=X+M
W(X)	
R(Y)	
	W(X)
Y=Y+N	
W(Y)	

T1	T2
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：

- R1(X)R2(X)W1(X)R1(Y)W2(X)W1(Y) ⇒ 不是冲突可串行化调度
- R1(X) W1(X)R2(X)W2(X)R1(Y) W1(Y) ⇒ 是冲突可串行化调度

- 优先图(precedence graph)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={ T1, T2,..., Tn }，Ti（I = 1, 2, ..., n）表示事务Ti对应的结点；
   - E为不同结点之间有向边的集合。
   - 有向边e_{ij}：Ti →Tj的定义如下：

     如果出现以下任一种情形，则结点Ti到结点Tj（i≠j）之间存在一条有向边

     – Wi(X)Rj(X)，Ri(X)Wj(X)，Wi(X)Wj(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?

- R1(X)R2(X)W1(X)R1(Y)W2(X)W1(Y)
- R1(X) W1(X)R2(X)W2(X)R1(Y) W1(Y)

- 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 r1 (X); r3 (X); w1(X); r2(X); w3(X)

② r3 (X); r2 (X); w3(X); r1(X); w1(X)

大纲

- 背景
- 并发控制概述
- 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
- 本章小结

两段锁协议

- 为了保证并发调度的正确性，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。
  - 之前的方法理论上可行，但实际运用中不可行
- 目前DBMS普遍采用两段锁协议（Two-phase Locking, 2PL）实现并发调度的可串行性，从而保证调度的正确性。
- 两段锁协议是指每个事务必须分为两个阶段对数据项进行加锁和解锁。
  - 在对任何数据进行读、写操作之前，首先要申请并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对任何其他封锁。
- 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。

- “两段”锁的含义是，事务分为两个阶段：
  - 第一阶段获得封锁，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Tj不遵守两段锁协议的封锁序列

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 A
Unlock B	
	W(A)=210
	Unlock C

- 事务T1的封锁序列：

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

- 事务T2的封锁序列：

Slock C　Xlock C　Slock A　Xlock A　Unlock C

调度L1

L1=R1(A)R2(C)W1(A)W2(C)R1(B)W1(B)R2(A)W2(A)

调度L2

L2=R1(A)W1(A)R1(B)W1(B)R2(C)W2(C)R2(A)W2(A)

可串行化调度

- 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 A
等待	等待

发生死锁

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 正确吗？）
- 两类可串行化调度的充分非必要条件
  - 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

- 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

大纲

- 背景
- 并发控制概述
- 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**
- 本章小结

封锁的粒度

- 封锁粒度（granularity）
  - 指封锁对象的大小
  - 粗粒度(coarse granularity) VS. 细粒度(fine granularity)
- 封锁的对象
  - 逻辑单元，物理单元
  - 关系数据库中的逻辑单元
    - 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
  - 关系数据库中的物理单元
    - 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，并发度较高，但系统开销也就越大。

[示例]：

① 若封锁粒度是数据页，事务T1需要修改元组L1，则T1必须对包含L1的整个数据页A加锁。如果T1对A加锁后事务T2要修改A中元组L2，则T2被迫等待，直到T1释放A。

② 如果封锁粒度是元组，则T1和T2可以同时对L1和L2加锁，不需要互相等待，提高了系统的并行度。

③ 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。

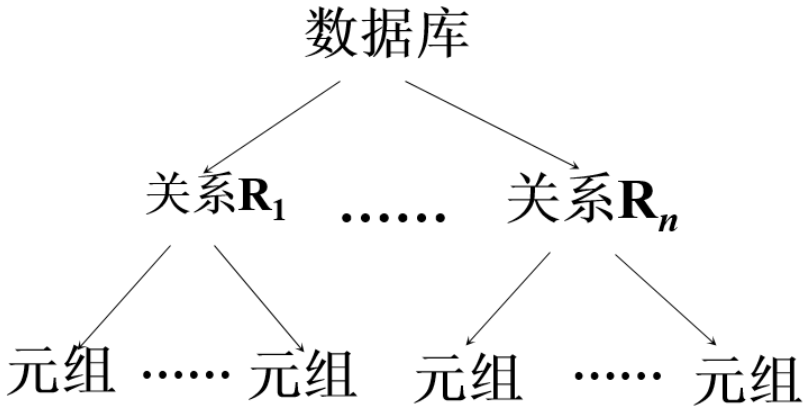
- 封锁粒度的选择
  - 选择封锁粒度时应同时考虑封锁开销和并发度两个因素，适当选择封锁粒度以求得最优效果。

一般考虑：

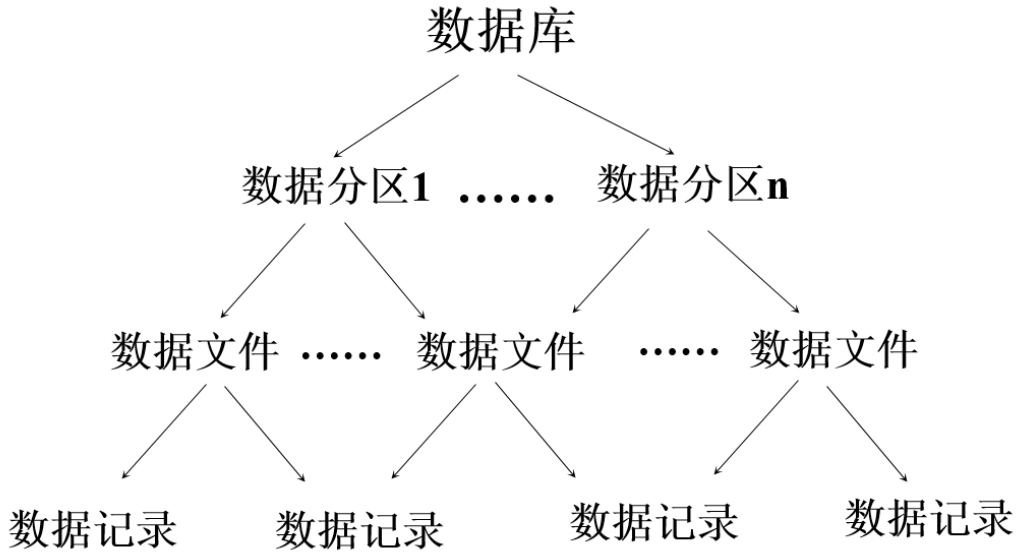
- 需要处理某个关系的大量元组的用户事务 — 以关系为封锁粒度
- 需要处理多个关系的大量元组的用户事务 — 以数据库为封锁粒度
- 只处理少量元组的用户事务 — 以元组为封锁粒度

- 多粒度封锁 (Multiple Granularity Locking)
  - 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Ti需要访问整个数据库，而且使用一种封锁协议，则事务Ti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Ti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Tj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，我们就可以构造出这样的一种机制。
  - 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。

- 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
  - 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
    - 显式封锁是应事务的要求直接加到数据对象上的锁；
    - 隐式封锁是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；
  - 显式封锁和隐式封锁的效果一样。

- 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R1加X锁，系统必须搜索其上级结点数据库、关系R1以及R1的下级结点，即R1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。
  - 特点：检查方法效率很低!
  - 解决方案：引入意向锁(intention lock)

- 意向锁
  - 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。
  - 有了意向锁，DBMS就无须逐个检查下一级结点的显式封锁。

- 引进意向锁的目的就是为了提高对某个数据对象加锁时系统的检查效率。
  - 假设事务Tk 希望封锁整个数据库，为此它需要对树的根结点加锁，但这种请求不会成功。因为当前Ti在树的某部分持有锁。但是，系统是怎样判定根结点是否可以加锁呢?

    方法1：搜索整棵树，但这种方法破坏了多粒度封锁机制的初衷。

    方法2：引入意向锁。在一个结点显式加锁之前，该结点的全部祖先结点均加上了意向锁。因此，事务判定是否能够成功给一个结点加锁时不必去搜索整棵树。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。

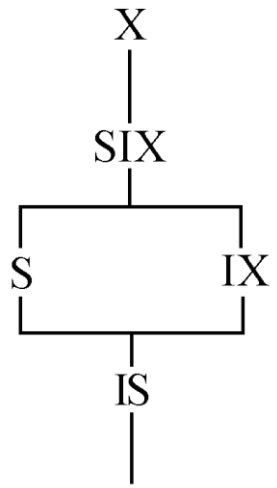
- 三种常用的意向锁
  - IS锁（意向共享锁, intent share lock）
    - 如果对一个数据对象加IS锁，表示它的后裔结点拟（意向）加S锁。
    - 例如：事务T1要对R1中某个元组加S锁，则要首先对关系R1和数据库加IS锁
  - IX锁（意向排他锁, intent exclusive lock）
    - 如果对一个数据对象加IX锁，表示它的后裔结点拟（意向）加X锁。
    - 例如：事务T1要对R1中某个元组加X锁，则要首先对关系R1和数据库加IX锁
  - SIX锁（共享意向排他锁, share intent exclusive lock）
    - 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



(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，则要首先对数据库加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)
- 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。

例题

- 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；R1上加IS锁；R2上加S锁；R3上加IS锁；r90到r100上加S锁，r201到r210上加S锁；

③ D上加IS锁和IX锁；R2上加SIX锁；

④ D上加IX锁；R1、R2、R3上加X锁。

课堂练习

- 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?

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、读“脏”数据三类问题
- 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

本章作业

- 教材第十一章习题：1-10，12-16.