数据库系统原理

第11章　并发控制

分级通关平台，QQ群

基础篇

第1章 绪论
第2章 关系数据库*3
第3章 标准语言SQL*3
第4章 数据库安全性
第5章 数据库完整性
实验1

设计与应用开发篇

第6章 关系数据理论*2
第7章 数据库设计
实验2
第8章 数据库编程

系统篇

第9章 *关系查询处理和优化
实验3
第10章 数据库恢复技术
第11章 并发控制
实验4

内容提要

- ✓ 并发控制概述
- ✓ 封锁
- ✓ 活锁和死锁
- ✓ 并发调度的可串行性
- ✓ 两段锁协议
- ✓ 封锁的粒度

并发控制

- 多用户数据库系统：允许多个用户同时使用
  - 飞机订票系统
  - 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
- 事务并发执行带来的问题
  - 会产生多个事务同时存取同一数据的情况
  - 可能会破坏事务隔离性和数据库的一致性

✓ 数据库管理系统必须提供并发控制机制

✓ 并发控制机制是衡量数据库管理系统性能的重要标志

并发控制

[例11.1]飞机订票系统中一个活动序列

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

④ 乙售票点也卖出一张机票，修改余额A←A-1，所以A为15

- 结果明明卖出两张机票，数据库中机票余额只减少1

	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

✓ 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，导致丢失修改。

并发控制

- 多事务执行方式

✓ 本章讨论单处理机系统为基础

1）事务串行执行

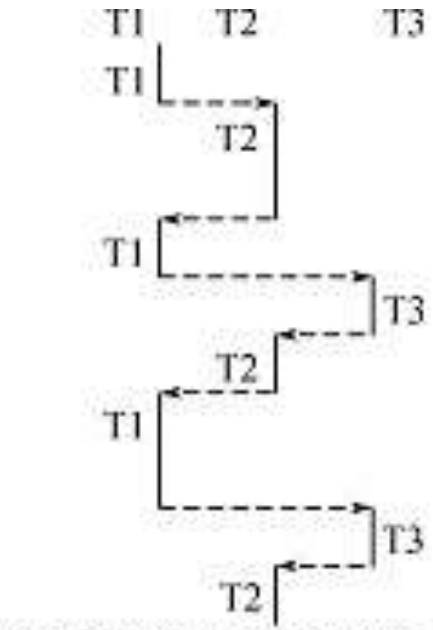
- 每个时刻只有一个事务运行，其他事务必须等待
- 不能充分利用系统资源，发挥共享资源的特点

2）交叉并发方式（Interleaved Concurrency）

- 单处理机系统中，并行事务在轮流交叉运行
- 能够减少处理机的空闲时间

3）同时并发方式（simultaneous concurrency）

- 多处理机系统中，可以同时运行多个并行事务
- 最理想的并发方式，但受制于硬件环境



(b) 事务的交叉并发执行方式

并发控制

- 并发操作带来的数据不一致性

1.丢失修改（Lost Update）

- 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

2.不可重复读（Non-repeatable Read）

- 不可重复读是指事务T_1读取数据后，事务T_2执行更新操作，使T_1无法再现前一次读取结果

3.读“脏”数据（Dirty Read）

- 事务T1修改某一数据，事务T2读取同一数据后，T1由于某种原因撤销，T1修改过的数据恢复原值，T2读到的数据就为“脏”数据

并发控制

1）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

2）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。

3）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。

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)	

- ✓ 不可重复读是指事务T_1读取数据后，事务T_2执行更新操作，使T_1无法再现前一次读取结果。

并发控制

- T1将C值修改为200，T2读到C为200
- T1由于某种原因撤销，其修改作废，C恢复原值100
- 这时T2读到的C为200，与数据库内容不一致，就是“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	

✓ **并发控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**

✓ 事务T1修改某数据，事务T2读取该数据后T1撤销使数据恢复原值，T2就遇到读“脏”数据。

并发控制

- 并发控制的主要技术
  - 封锁(Locking)
  - 时间戳(Timestamp)
  - 乐观控制法
  - 多版本并发控制(MVCC)

- 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

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

封锁

- **排它锁又称为写锁X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
- **共享锁又称为读锁S锁**
  - 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

封锁--锁的相容矩阵

- T_2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T_2的封锁要求与T_1已持有的锁相容，封锁请求可以满足
  - N表示T_2的封锁请求与T_1已持有的锁冲突，T_2的请求被拒绝

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

封锁—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，称为封锁协议（Locking Protocol）。
  - **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
  1. 一级封锁协议
  2. 二级封锁协议
  3. 三级封锁协议

封锁—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）

	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

- 一级封锁协议可防止丢失修改，并保证事务T是可恢复的。

① 事务T_1在读A进行修改之前先对A加X锁

② 当T_2再请求对A加X锁时被拒绝

③ T_2只能等待T_1释放A上的锁后获得对A的X锁

④ 这时T_2读到的A已经是T_1更新过的值15

⑤ T_2按此新的A值进行运算，并将结果值A=14写回到磁盘。没有丢失修改

封锁—封锁协议

- 二级封锁协议
  - 一级封锁协议+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。
- 二级封锁协议可以防止丢失修改和读“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。

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

① 事务T1在对C进行修改之前，先对C加X锁，修改其值

② T2请求在C上加S锁，因T1已在C上加了X锁，T2等待

③ T1因某种原因被撤销，C恢复为原值100

④ 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

封锁—封锁协议

- 三级封锁协议
  - 一级封锁协议+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。

➢ 三级封锁协议可防止丢失修改、读脏数据和不可重复读。

① 事务T1在读A，B之前，先对A，B加S锁

② 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

③ 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

④ 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

⑤ T1结束才释放A，B上的S锁。T2才获得对B的X锁

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

封锁—封锁协议

- 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

	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		√		√	√	√	√

活锁与死锁

- 封锁可以解决并行操作的一致性问题，但也存在问题

- 事务T_1封锁了数据R
- 事务T_2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

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	•	•
			•

T有可能一直等待，这就是活锁的情形

活锁与死锁

- 封锁可以解决并行操作的一致性问题，但也存在问题
- 事务T1封锁了数据R1
- 事务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
	•
	•
	•

T1和T2两个事务永远不能结束，形成死锁

活锁与死锁

- 封锁技术可能存在活锁与死锁问题
  - ✓ 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构造事务队列
    - 该数据对象上的锁一旦释放，首先批准申请队首事务获得锁
  - ✓ 预防死锁的方法：破坏产生死锁的条件

1）一次封锁法

- ✓ 要求每个事务一次将要使用的数据全部加锁，否则不能继续执行

2）顺序封锁法

- ✓ 预先对数据对象规定一个封锁顺序，都按这个顺序实行封锁

封锁

- 一次封锁法：难于事先精确确定封锁对象，降低并发度
  - 数据库中数据是不断变化的，在执行过程中会增加封锁对象
  - 若将可能要封锁的数据对象全部加锁，将进一步降低并发度
- 顺序封锁法：维护成本高，也难以实现
  - 需要封锁的数据对象极多，并且随数据的增/删等操作不断变化
  - 随着事务的执行，很难事先确定每一个事务要封锁哪些对象
- 结论
  - 这两种预防死锁的策略在实际运行的很多情况下不太适合
  - 在解决死锁的问题上，实际普遍采用的是**诊断并解除死锁**的方法

封锁

- 超时法诊断死锁
  - 如果一个事务的等待时间超过了时限，就认为死锁了
  - 问题：
    - **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**
- 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

并行调度的可串行性

- 对并发事务不同的调度可能会产生不同的结果
  - 串行调度是正确的，执行结果等价于串行调度的调度也是正确的，称为**可串行化调度**
  - ✓ 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时，对多个事务的并发执行才是正确的
  - ✓ 一个给定的并发调度，当且仅当它是可串行化的，才是正确调度

[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

请给出对这两个事务不同的调度策略

并行调度的可串行性

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=3，B=4

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	

A=4，B=3

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

A=3，B=4

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

A=3，B=3

并行调度的可串行性

- 冲突操作：是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，可串行调度
  - 不能交换（Swap）的动作：不同事务的冲突操作

[例11.3] 有调度

$Sc_1=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$

$Sc_2=r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)$

T_1　　T_2

✓ Sc_2等价于一个串行调度T_1，T_2，所以Sc_1是冲突可串行化的调度

并行调度的可串行性

- 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
- [例11.4]有3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
  - ✓ 但是调度L_2是可串行化的，因为L_2执行的结果与L_1相同
  - ✓ Y的值都等于T_2的值，X的值都等于T_3的值
  - 存在不满足冲突可串行化条件的可串行化调度

两段锁协议2PL

- 普遍采用两段锁协议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
  - 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
  - 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
      - 第二阶段是释放封锁，也称为收缩阶段

两段锁协议

- 事务T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　→|

- 事务T不遵守两段锁协议，其封锁序列是：

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

- 右图的调度是遵守两段锁协议的。
- 如何验证T是一个可串行化调度？

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

两段锁协议

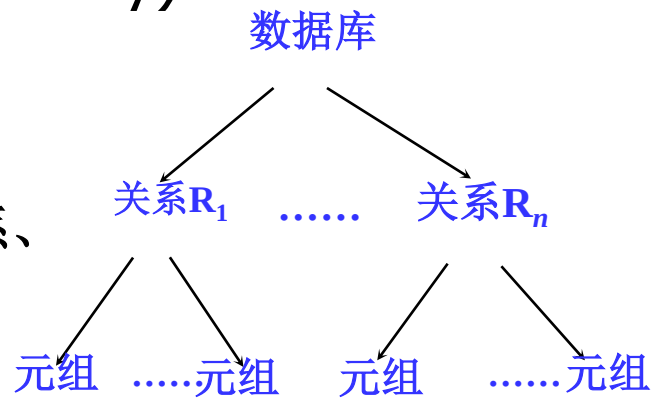
- 事务遵守两段锁协议是可串行化调度的充分条件，而不是必要条件。
  - 若并发事务都遵守两段锁协议，则对这些事务的任何并发调度策略都是可串行化的
  - 若并发事务的一个调度是可串行化的，不一定所有事务都符合两段锁协议
- 两段锁协议与防止死锁的一次封锁法
  - 一次封锁法要求每个事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议
  - 两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此可能发生死锁

封锁的粒度

- 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
- 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



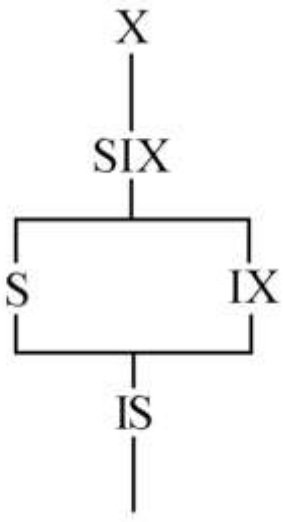
封锁的粒度—多粒度封锁

- 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
  - 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
- 系统检查封锁冲突时
  - 要检查显式封锁
  - 还要检查隐式封锁
- 加锁时，系统要检查
  - 该数据对象
  - 所有上级结点
  - 所有下级结点？

封锁的粒度—意向锁

- 引进意向锁(intention lock)，提高加锁时系统的检查效率
  - 如果对一个结点加意向锁，则说明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
- 意向共享锁(Intent Share Lock，简称IS锁)
  - 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
- 意向排它锁(Intent Exclusive Lock，简称IX锁)
  - 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
- 共享意向排它锁(Share Intent Exclusive Lock，简称SIX锁)
  - 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）

封锁的粒度—意向锁



- 锁的强度
  - 锁的强度是指它对其他锁的排斥程度
  - 申请封锁时以强锁代替弱锁是安全的，反之不然
- 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（多粒度树）
  - 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)

封锁的粒度—意向锁

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

- 具有意向锁的多粒度封锁方法
  - 提高了系统的并发度，得到广泛应用
  - 减少了加锁和解锁的开销

小结

- 并发控制通常使用封锁机制
  - ✓ 排它锁、共享锁
  - ✓ 三级封锁协议
- 活锁与死锁
- 并发调度的可串行性
  - ✓ 冲突可串行化调度
  - ✓ 两段封锁协议
- 封锁的粒度
  - ✓ 意向锁
  - ✓ 意向锁是由数据引擎自己维护的