



第十一章 并发控制

Concurrency Control

教师：林友芳





Concurrent vs Parallel Computing

► Parallel computing

- is a type of computation in which many calculations or the execution of processes **are carried out simultaneously**.
- **Large** problems can often be **divided into smaller ones**, which can then be **solved at the same time**.

► Concurrent computing

- Is a form of computing in which several computations are executed **during overlapping time periods**—concurrently—instead of sequentially
- The goal here is to model processes **in the outside world that happen concurrently**, such as multiple clients accessing a server at the same time.

► Concurrent processes can be executed on one core by

interleaving the execution steps of each process via time-sharing slices



问题的产生

1. 多用户数据库系统的存在

允许多个用户同时使用的数据库系统

➤ 飞机定票数据库系统

➤ 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个



问题的产生 (续)

不同的多事务执行方式

(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

T1

T2

T3

事务的串行执行方式



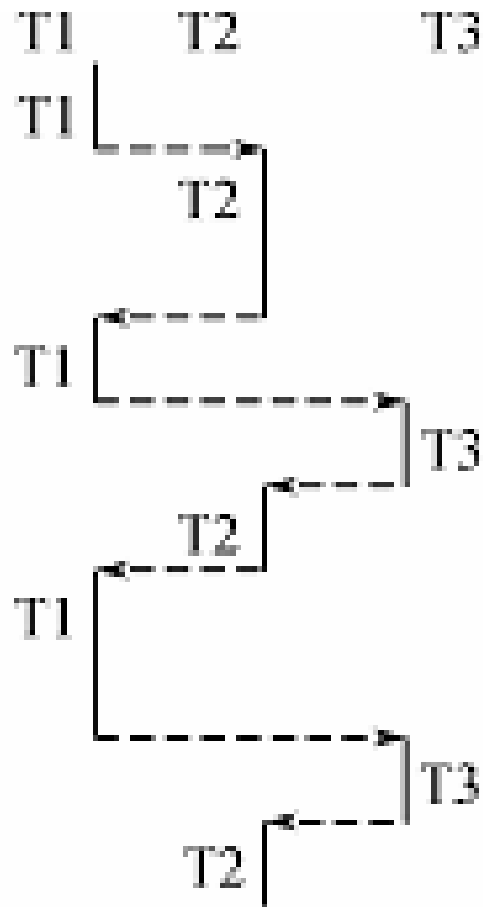
问题的产生 (续)

(2) 交叉并发方式 (Interleaved Concurrency)

- 在单处理机系统中，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
- 单处理机系统中的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



问题的产生 (续)



事务的交叉并发执行方式



问题的产生 (续)

(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**并发事务真正的并行运行**



问题的产生 (续)

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
- 可能会存取和存储不正确的数据，**破坏事务一致性和数据库的一致性**



并发控制内容列表

11.1 并发控制概述

11.2 封锁

11.3 活锁和死锁

11.4 并发调度的可串行性

11.5 两段锁协议

11.6 封锁的粒度

11.7 小结



11.1 并发控制概述

■ 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



并发操作带来数据的不一致性实例

[例1]飞机订票系统中的一个活动序列

- ① 甲售票点(甲事务)读出某航班的机票余额A, 设 $A=16$;
 - ② 乙售票点(乙事务)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 $A \leftarrow A-1$, 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 $A \leftarrow A-1$, 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



并发操作带来数据的不一致性实例

- 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，对甲、乙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，甲事务的修改就被丢失。

原因：第4步中乙事务修改A并写回后覆盖了甲事务的修改



并发控制概述（续）

■ 并发操作带来的数据不一致性

- 丢失修改 (Lost Update)
- 不可重复读 (Non-repeatable Read)
- 读“脏”数据 (Dirty Read)

■ 记号

- $R(x)$: 读数据 x
- $W(x)$: 写数据 x



1. 丢失修改

- 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。
- 上面飞机订票例子就属此类



丢失修改 (续)

T_1	T_2
① $R(A)=16$	
②	$R(A)=16$
③ $A \leftarrow A-1$ $W(A)=15$	
④	$A \leftarrow A-1$ $W(A)=15$

丢失修改



2. 不可重复读

- 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。
- 不可重复读包括三种情况：
 - (1) 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



不可重复读 (续)

例如:

T_1	T_2
① $R(A)=50$ $R(B)=100$ 求和=150	
②	$R(B)=100$ $B \leftarrow B * 2$ $W(B)=200$
③ $R(A)=50$ $R(B)=200$ 和=250 (验算不对)	

- T1读取B=100进行运算
- T2读取同一数据B, 对其进行修改后将B=200写回数据库。
- T1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



不可重复读 (续)

(2)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，发现某些记录消失了

(3)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**
(Phantom Row)



3. 读“脏”数据

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



读“脏”数据 (续)

例如

T_1	T_2
① $R(C)=100$ $C \leftarrow C*2$ $W(C)=200$	
②	$R(C)=200$
③ ROLLBACK C恢复为100	

- T_1 将C值修改为200, T_2 读到C为200
- T_1 由于某种原因撤销, 其修改作废, C恢复原值100
- 这时 T_2 读到的C为200, 与数据库内容不一致, 就是“脏”数据



并发控制概述

- **数据不一致性**：由于**并发操作破坏了事务的隔离性**
- **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



并发控制概述（续）

■ 并发控制的主要技术

- 有封锁(Locking)
- 时间戳(Timestamp)
- 乐观控制法

■ 商用的DBMS一般都采用封锁方法



11.2 封锁

- 什么是封锁
- 基本封锁类型
- 锁的相容矩阵



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，其它的事务不能更新此数据对象。**



基本封锁类型

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



排它锁

- 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
- 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

- 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
- 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

$T_2 \backslash T_1$	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（续）

在锁的**相容**矩阵中：

- 最左边一列表示事务 T_1 已经获得的数据对象上的锁的类型，其中横线表示没有加锁。
- 最上面一行表示另一事务 T_2 对同一数据对象发出的封锁请求。
- 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



使用封锁机制解决丢失修改问题

T_1	T_2	没有丢失修改
① Xlock A		
② $R(A)=16$		■ 事务 T_1 在读A进行修改之前先对A加X锁
③ $A \leftarrow A-1$	Xlock A	■ 当 T_2 再请求对A加X锁时被拒绝
$W(A)=15$	等待	■ T_2 只能等待 T_1 释放A上的锁后 T_2 获得对A的X锁
Commit	等待	■ 这时 T_2 读到的A已经是 T_1 更新过的值15
Unlock A	等待	■ T_2 按此新的A值进行运算，并将结果值 $A=14$ 送回到磁盘。避免了丢失 T_1 的更新。
④	获得Xlock A	
	$R(A)=15$	
	$A \leftarrow A-1$	
⑤	$W(A)=14$	
	Commit	
	Unlock A	

使用封锁机制解决不可重复读问题

可重复读

- 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 T_1 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

T_1	T_2
① Slock A	
Slock B	
R(A)=50	
R(B)=100	
求和=150	
②	Xlock B
	等待
	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
	$B \leftarrow B * 2$
⑤	W(B)=200
	Commit
	Unlock B



使用封锁机制解决读“脏”数据

T_1	T_2
① Xlock C $R(C)=100$ $C \leftarrow C * 2$ $W(C)=200$	
②	Slock C 等待
③ ROLLBACK (C恢复为100) Unlock C	等待 等待 等待
④	获得Slock C $R(C)=100$
⑤	Commit C Unlock C

不读“脏”数据

- 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 读“脏”数据



11.3 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



11.3.1 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 有可能永远等待或等待时间过长, 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	.
Unlock	等待	.	Lock R
.	等待	Lock R	等待
.	等待	.	等待
.	等待	Unlock	等待
.	等待	.	Lock R
.	等待	.	.

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



11.3.2 死锁

- 事务 T_1 封锁了数据 R_1
- T_2 封锁了数据 R_2
- T_1 又请求封锁 R_2 ，因 T_2 已封锁了 R_2 ，于是 T_1 等待 T_2 释放 R_2 上的锁
- 接着 T_2 又申请封锁 R_1 ，因 T_1 已封锁了 R_1 ， T_2 也只能等待 T_1 释放 R_1 上的锁
- 这样 T_1 在等待 T_2 ，而 T_2 又在等待 T_1 ， T_1 和 T_2 两个事务永远不能结束，形成死锁



死锁 (续)

T_1	T_2
lock R_1	•
•	Lock R_2
•	•
Lock R_2 .	•
等待	•
等待	Lock R_1
等待	等待
等待	等待
	•
死	锁



解决死锁的方法

两类方法

1. 预防死锁
2. 死锁的诊断与解除



1. 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



死锁的预防 (续)

预防死锁的方法

- 一次封锁法
- 顺序封锁法



(1)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
- 存在的问题
 - 降低系统并发度
 - **难于事先精确确定封锁对象**



(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，并且在不断地变化。
 - 难以实现
很难事先确定每一个事务要封锁哪些对象



死锁的预防 (续)

■ 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



2. 死锁的诊断与解除

死锁的**诊断**

- 超时法

- 事务等待图法



(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—认定为死锁**

➤ **优点：实现简单**

➤ **缺点**

– 太短有可能**误判死锁**

– 时限若设置得太长，死锁发生后不能及时发现



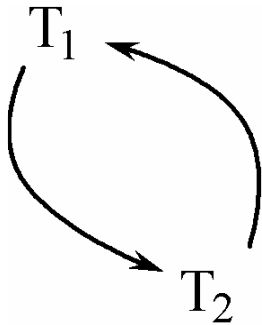
(2)等待图法

用事务等待图动态反映所有事务的等待情况

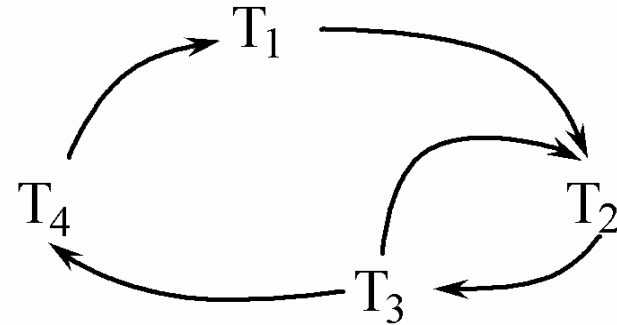
- 事务等待图是一个有向图 $G = (T, U)$
- T 为结点的集合，每个结点表示正运行的事务
- U 为边的集合，每条边表示事务等待的情况
- 若 T_1 等待 T_2 ，则 T_1, T_2 之间划一条有向边，从 T_1 指向 T_2



等待图法 (续)



(a)



(b)

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—**消灭回路**

- **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



11.4 并发事务调度的可串行化

- ▶ **A schedule or a timetable**, as a basic time-management tool, consists of a list of times at which possible tasks, events, or actions are intended to take place, or of a **sequence of events** in the chronological order in which such things are intended to take place.
- ▶ The process of **creating a schedule** - deciding how to order these tasks and how to commit resources between the variety of possible tasks - is called **scheduling**, and a person responsible for making a particular schedule may be called a **scheduler**.
- ▶ **调度**—scheduling, dispatching



并发调度的结果

- **DBMS对并发事务不同的调度可能会产生不同的结果**
- **问题：什么样的调度是正确的？**



11.4.1 可串行化调度

► 1. 可串行化(Serializable)调度

- 多个**并发事务的并行执行**是正确的，当且仅当其结果与按**某一次序串行地**执行这些事务时的结果相同
- 给定调度方案的并行执行结果与某一串行执行结果等价

► 2. 可串行性(Serializability)准则

- 是并发事务正确调度的准则
- 一个给定的**并发事务调度方案**，当且仅当它是可串行化的，才认为是正确调度
- 如何判定一个并发调度方案能否转化为某一等价的串行调度



例

- ▶ 设有两个事务，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
 - 现给出对这两个事务不同的调度策略



串行化调度 (a) , 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

- 假设A、B的初值均为2。
- 按 $T_1 \rightarrow T_2$ 次序执行结果为 $A=3, B=4$
- 串行调度策略,正确的调度



串行化调度(b), 正确的调度

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	

■ 假设A、B的初值均为2。

■ $T_2 \rightarrow T_1$ 次序执行结果为

$B=3, A=4$

■ 串行调度策略,正确的调度



不可串行化调度，错误的调度

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

- 该调度方案执行结果与两个串行执行方案(a)和(b)的结果都不同
- 是错误的调度
- 如何判定？



可串行化调度，正确的调度

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

- 执行结果与串行调度(a)的执行结果相同
- 是正确的调度
- 如何判定？



11.4.2 冲突可串行化调度

► 冲突操作

► 冲突操作是指不同的事务对同一个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 */

– $W_i(x)$ 与 $W_j(x)$ /* 事务 T_i 写 x , T_j 写 x */

► 其他操作是不冲突操作



并发事务操作次序的可交换性

- 调度序列中**操作次序的可交换性**
 - 交换给定调度序列中的**相邻操作**（同一事务或不同事务的）**是否会影响**并发调度的**结果**
 - 会影响则不可交换，不会影响则可交换
- 结论：给定调度序列中
 - 不同事务的**冲突操作**不能交换(Swap)
 - 同一事务的**两个操作**不能交换(Swap)



一个调度可串行化的充分条件

➤ 教材说法:

- 一个调度 S_c 在保证**冲突操作**的**次序不变**的情况下, 通过多次**交换两个事务不冲突操作的次序**得到另一个调度 S_c' , 如果 S_c' 是串行的, 称调度 S_c 为冲突可串行化的调度

➤ 更严谨的说法:

- 一个调度 S_c 在满足前后相邻操作的可交换性的前提下, 通过迭代**交换 S_c 中属于不同事务的相邻不冲突操作的次序**得到另一个调度 S_c' , 如果 S_c' 是串行的, 称调度 S_c 为**冲突可串行化**的调度
- 一个调度是**冲突可串行化**, 一定是**可串行化**的调度



调度串行化案例

设有调度

$T_1: r(A) \ w(A) \qquad r(B) \ w(B)$

$T_2: \qquad r(A) \ w(A) \qquad r(B) \ w(B)$

$Sc_1 = r_1(A)w_1(A)r_2(A)\mathbf{w_2(A)r_1(B)w_1(B)}r_2(B)w_2(B)$

把 $w_2(A)$ 与 $r_1(B)w_1(B)$ 交换, 得到:

$r_1(A)w_1(A)\mathbf{r_2(A)r_1(B)w_1(B)}w_2(A)r_2(B)w_2(B)$

再把 $r_2(A)$ 与 $r_1(B)w_1(B)$ 交换:

$Sc_2 = \mathbf{r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)}$

- Sc_2 等价于一个串行调度 T_1, T_2 ,
- Sc_1 是冲突**可串行化**的调度



可串行化调度的条件

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。还有不满足冲突可串行化条件的可串行化调度。

[例]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



问题

- ▶ 在实际调度中数据库系统需要去判定的一个调度是否可串行化吗？
- ▶ 存在不存在把并发事务的调度方案串行化的问题或需要？
- ▶ 如果不存在不需要，真正需要的东西是什么？
- ▶ 真正需要的东西
 - 数据库管理系统的并发控制系统有没有一定的机制或手段来保证并发事务调度结果的可串行化
 - 换言之：有没有机制，使得只要写事务的程序员遵循一定的原则，就可以使得并发事务调度变得简单，可以完全不必考虑如何细心地安排操作的执行次序，并保证结果的可串行化



11.5 两段锁协议

■ 封锁协议

运用封锁方法时，对数据对象加锁时需要约定一些规则

- **何时申请封锁**
- **持锁时间**
- **何时释放封锁等**

- **两段封锁协议(Two-Phase Locking, 简称2PL)是最常用的一种封锁协议，理论上证明使用两段封锁协议产生的是可串行化调度**



11.5 两段锁协议

■ 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**



11.5 两段锁协议

■ “两段” 锁的含义

从锁与解锁的阶段角度，将事务分为两个阶段

1、第一阶段是**获得封锁**，也称为扩展阶段

➤ 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

2、第二阶段是**释放封锁**，也称为收缩阶段

➤ 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



11.5 两段锁协议

例：

➤ **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 →|

➤ **事务 T_j 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;



两段锁协议 (续)

事务T1	事务T2
Slock(A) R(A=260)	Slock(C) R(C=300)
Xlock(A) W(A=160)	Xlock(C) W(C=250) Slock(A) 等待 等待 等待 等待 等待 R(A=160) Xlock(A)
Slock(B) R(B=1000) Xlock(B) W(B=1100) Unlock(A)	W(A=210) Unlock(C) Unlock(A)
Unlock(B)	

- 左图的调度是遵守两段锁协议的，因此一定是一个可串行化调度。

遵守两段锁协议的可串行化调度



两段锁协议 (续)

- 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**—易于并发控制
- 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 (续)

■ 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**
- **一次封锁法更粗暴——牺牲效率和并发性**



两段锁协议 (续)

[例] 遵守两段锁协议的事务发生死锁

T_1	T_2
Slock B $R(B)=2$	
	Slock A $R(A)=2$
Xlock A 等待 等待	Xlock B 等待

遵守两段锁协议的事务可能发生死锁



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——**简单粗暴成本低效果差**
 - 封锁的粒度越小，并发度较高，但系统开销也就越大——**复杂精细成本高效果好**



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 可以同时分别对 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



多粒度封锁

■ 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



多粒度封锁

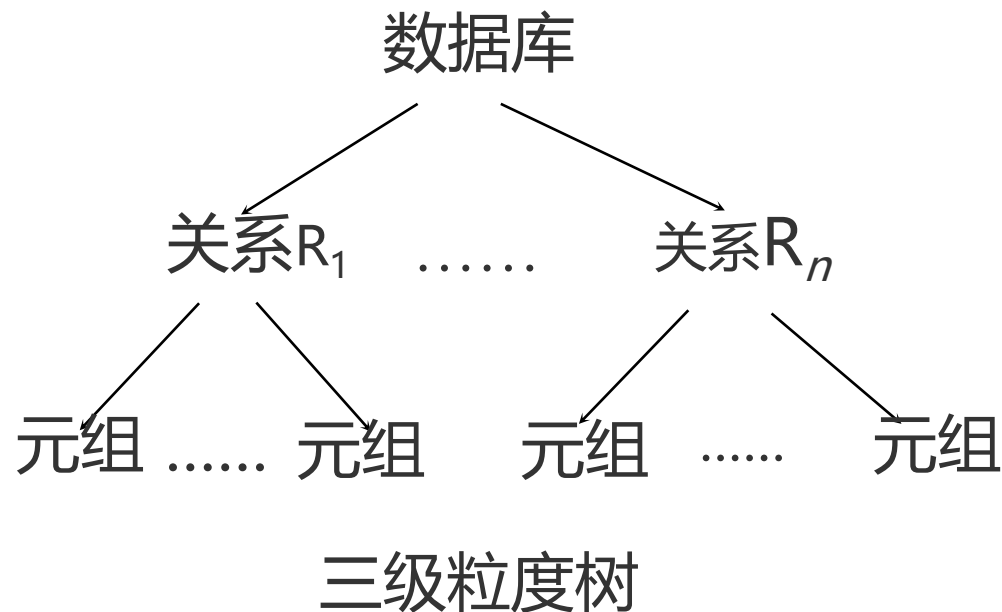
■ 多粒度树

- 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。





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
 - 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:

- 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



加锁时的封锁检查

► 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



加锁时的封锁检查

► 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**

看结点上的显式封锁是否与由本事务带来的对结点的隐式封锁将加到下级结点的封锁）冲突



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**
- 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
- 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**



小结 (续)

- **并发控制机制产生的并发事务调度操作结果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，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



本章作业

P.306(第四版)

9,10

P.326 (第五版)

9,10