实现篇
第十章 并发控制

主讲：程思瑶
海量数据计算研究中心

目录

- 事务概念
- 事务的并发执行和调度
- 并发控制协议

目录

- 事务概念
- 事务的并发执行和调度
- 并发控制协议

事务概念

- 事务(Transaction)
  - 是用户定义的一个数据库操作序列，这些操作要么全做，要么全不做，是一个不可分割的工作单位。
- 事务与程序不同
  - 在关系数据库中，一个事务可以是一条SQL语句，一组SQL语句或整个程序
  - 一个程序通常包含多个事务
- 事务是并发控制和恢复的基本单位

- 定义事务
  - 显式定义方式

BEGIN TRANSACTION
SQL 语句1
SQL 语句2
。。。。。
COMMIT

BEGIN TRANSACTION
SQL 语句1
SQL 语句2
。。。。。
ROLLBACK

- 事务异常终止
- 事务运行的过程中发生了故障，不能继续执行
- 系统将事务中对数据库的所有已完成的操作全部撤销
- 事务滚回到开始时的状态

（读+更新）

的更新写回到磁盘

DBMS按缺省规定自动划分事务

事务概念

- 事务的特性（ACID）
  - 原子性 (Atomicity)：即事务完全执行或完全不执行
  - 一致性(Consistency)：事务执行的结果必须是使数据库从一个一致性状态变到另一个一致性状态。
    - 一致性状态：
      - 数据库中只包含成功事务提交的结果
    - 不一致性状态：
      - 数据库系统运行中发生故障，有些事务尚未完成就被迫中断，这些未完成事务对数据库所做的修改有一部分已写入物理数据库，这时数据库就处于一种不正确的状态

- 一致性与原子性

银行转帐：从帐号A中取出一万元，存入帐号B。

– 定义一个事务，该事务包括两个操作

A	B
A=A-1	B=B+1

– 这两个操作要么全做，要么全不做

- 全做或者全不做，数据库都处于一致性状态。
- 如果只做一个操作，用户逻辑上就会发生错误，少了一万元，数据库就处于不一致性状态。

事务概念

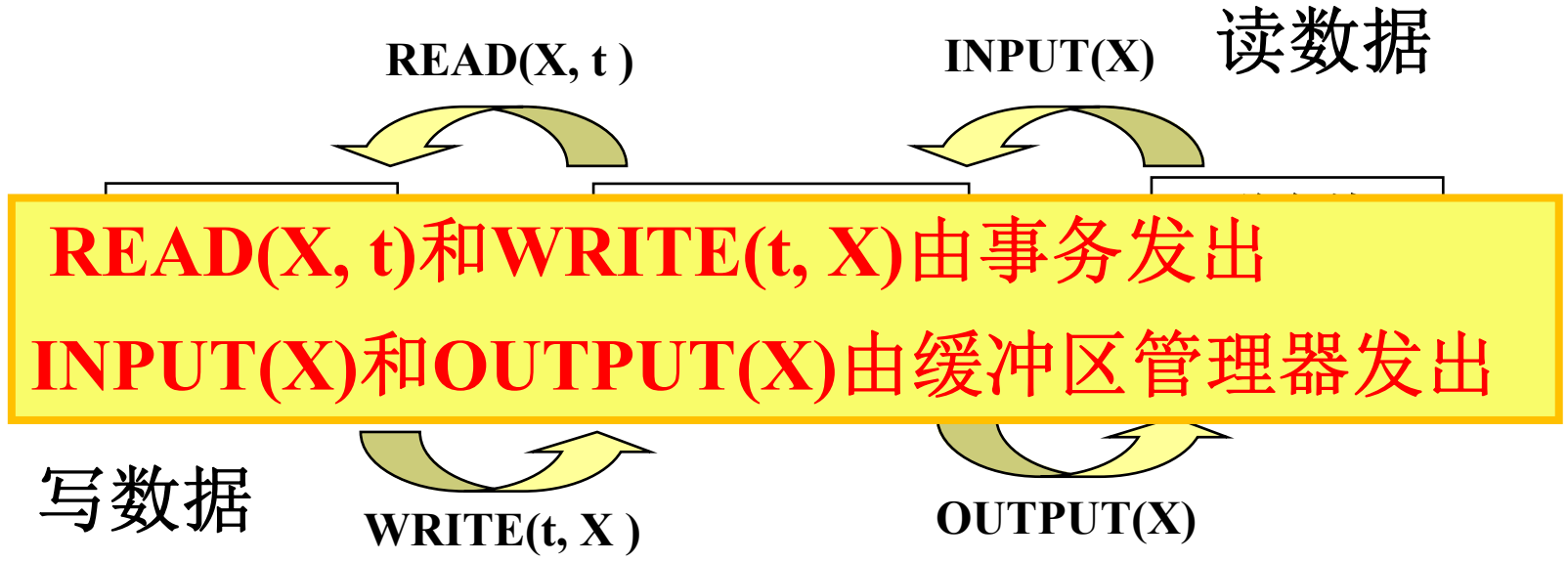
- 事务的特性（ACID）
  - 隔离性(Isolation)：表面看起来，每个事务都是在没有其它事务同时执行的情况下执行的。
    - 一个事务内部的操作及使用的数据对其他并发事务是隔离的
    - 并发执行的各个事务之间不能互相干扰
  - 持久性(Durability)：一个事务一旦提交，它对数据库中数据的改变就应该是永久性的
    - 接下来的其他操作或故障不应该对其执行结果有任何影响

事务概念

- 保证事务ACID特性是事务处理的任务
- 破坏事务ACID特性的因素
  - 多个事务并行运行时，不同事务的操作交叉执行
    - 数据库管理系统必须保证多个事务的交叉运行不影响这些事务的隔离性
  - 事务在运行过程中被强行停止
    - 数据库管理系统必须保证被强行终止的事务对数据库和其他事务没有任何影响

事务概念

- 事务运用以下两个操作访问数据
  - Read(X):
    - 从数据库把数据项X传送到事务的局部缓冲区
  - Write(X)
    - 从事务的局部缓冲区把数据项X传回数据库



目录

- 事务概念
- **事务的并发执行和调度**
- 并发控制协议

事务的并发执行和调度

- 调度
  - 一个或多个事务的重要操作按时间排序的一个序列
  - READ, WRITE序列
- 串行调度
  - 如果一个调度S的动作组成首先是一个事务的所有动作，然后是另一个事务的所有动作，依此类推、没有动作的混合，那么我们说这一调度是串行的。
    - 更精确地讲，如果有任意两个事务T和T’，若T的某个动作在T’的某个动作前，则T的所有动作在T’的所有动作前，那么调度S是串行的。

Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.

串行调度

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

串行调度1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

等价串行调度1的一个并发调度

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

一个违反一致性的并发调度

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

如何保证一个调度在某种意义上等价于一个串行调度？

事务的并发执行和调度

- 可串行化调度
  - 如果不管数据库初始状态如何，一个调度对数据库状态的影响都和某个串行调度相同，则我们说这个调度是可串行化的。

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t+100			
WRITE(B,t)			125
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

图9-5 一个非串行的可串行化调度

T_1	T_2	A	B
		25	25
READ(A,t)			
t := t+100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		50
READ(B,t)			
t := t+100			
WRITE(B,t)			150

图9-6 一个非可串行化的调度

事务的并发执行和调度

- 事务和调度的一种记法
  - 只考虑事务的读写操作
  - 用$r_i(X)$、$w_i(X)$表示事务T_i读和写数据库元素X

    T1：r1(A)，w1(A)，r1(B)，w1(B)

    T2 ：r2(A)，w2(A)，r2(B)，w2(B)

  - 事务集合T的调度S是组成它的事务动作的一个交错的序列

r1(A)，w1(A)，　r2(A)，w2(A)，r1(B)，w1(B)，r2(B)，w2(B)

事务的并发执行和调度

- 冲突
  - 调度中一对连续的动作，它们满足：如果它们的顺序交换，那么涉及的事务中至少有一个的行为会改变。

例如，设T_i、T_j是不同的事务，即$i \neq j$。

- 同一事务的两个动作是冲突的，如$r_i(X);w_i(Y)$。原因在于单个事务的动作顺序是固定的，不能被DBMS重新排列；
- 不同事务对同一数据库元素的写冲突。即$w_i(X);w_j(X)$是一个冲突。
- 不同事务对同一数据库元素的读和写也冲突。即

不同事务的任何两个动作在顺序上可以交换，除非

- 它们涉及同一数据库元素，并且
- 至少有一个动作是写

事务的并发执行和调度

- 冲突可串行化
  - 我们说两个调度是冲突等价的，如果通过一系列相邻动作的非冲突交换能将它们中的一个转换为另一个。
  - 如果一个调度冲突等价于一个串行调度。那么我们说该调度是冲突可串行化的

考虑如下调度：

r1(A)，w1(A)，　r2(A)，w2(A)，r1(B)，w1(B)，r2(B)，w2(B)

我们说这个调度是冲突可串行化的。

将这一调度转换为串行调度（T_1，T_2）

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

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

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

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

$r_1(A);\ w_1(A);\ r_1(B);\ w_1(B);\ r_2(A);\ w_2(A);\ r_2(B);\ w_2(B);$

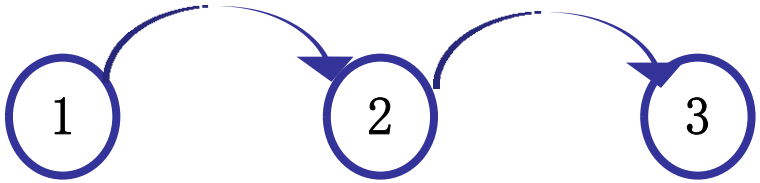
图9-8　通过交换相邻动作将冲突可串行化调度转换为串行调度

事务的并发执行和调度

- 优先关系
  - 已知调度S，其中涉及事务T1和T2，可能还有其它事务。我们说T1优先于T2，记作T1<T2，如果有T1的动作A1和T2的动作A2，满足：
    - 在S中A1在A2前；
    - A1和A2都涉及同一数据库元素；并且
    - A1和A2中至少有一个动作是写。
  - 因此，在任何冲突等价于S的调度中，A1将出现在A2前。所以，如果这些调度中有一个是串行调度，那么该调度必然使T1在T2前。

- 优先图
  - 优先图的节点是调度S中的事务。当这些事务是具有不同的i的Ti时，我们将仅用整数i来表示Ti的结点。如果Ti<Tj，则有一条从结点i到结点j的弧。
  - 例如，有调度S：

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$



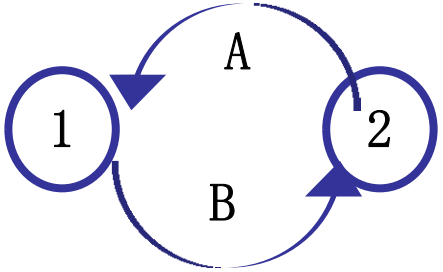
调度顺序为：T1→T2→T3

- 冲突可串行性判断
  - 构造调度S的优先图，判断其中是否有环。
  - 例1，调度S是无环的：

    $r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$
  - 例2，考虑调度S'：

    $r_2(A); r_1(B); w_2(A); r_2(B); r_1(A); w_1(B); w_1(A); w_2(B);$

构造对应的优先图，



目录

- 事务概念
- 事务的并发执行和调度
- **并发控制协议**

并发控制协议

- 基于锁的协议
- 基于时间戳的协议
- 多版本机制
- 快照隔离
- 等等….

并发控制协议

- 基于锁的协议
- 基于时间戳的协议
- 快照隔离
- 多版本机制
- 等等….

基于锁的并发控制协议

- 锁的概念
- 死锁、死锁的处理
- 两段锁并发控制协议

基于锁的并发控制协议

- 锁的概念
- 死锁、死锁的处理
- 两段锁并发控制协议

锁的概念

- 一个保证可串行性的方法是在互斥的方式下存取数据项，即当一个事务存取一个数据项时不允许其他事务修改这个数据项。
- 可以通过基于锁的并发控制协议实现。

锁的概念

- 锁的概念
  - 锁是数据项上的并发控制标志。锁可以分为两种类型：
    - (1) 共享锁 如果事务T得到了数据项Q上的共享锁，则T可以读这个数据项，但不能写这个数据项。共享锁表示为S。
    - (2) 互斥锁 如果事务T得到了数据项Q上的互斥锁，则T既可以读这个数据项，也可以写这个数据项。互斥锁表示为X。

锁的概念

- 锁的概念
  - 每个事务在存取一个数据项之前必须获得这个数据项上的锁。
  - 一个事务需要获得的锁的类型依赖于它将在数据项上执行什么样的操作。
  - 给定一个各种类型锁的集合，如下定义这个锁集合上的相容关系。
    - 令A和B表示任意类型的锁。设事务T_i在数据项Q上要求一个A型锁，事务T_j ($T_i \neq T_j$)已经在Q上有一个B型锁。如果事务T_i能够获得Q上的A型锁，则说A型锁和B型锁是相容的。
    - 锁之间的相容关系可以使用一个矩阵来表示。矩阵的元素COMP(A, B)=true当且仅当A型锁与B型锁是相容的。

	S	X
S	true	false
X	false	false

锁的概念

- 锁的概念
  - 事务通过执行LOCK-S(Q)操作申请数据项Q上的共享锁。
  - 事务通过执行LOCK-X(Q)操作申请数据项Q上的互斥锁。
  - UNLOCK(Q)操作用来释放数据项Q上的锁。
  - 如果事务T要存取数据项Q，T必须申请在Q上加锁。如果数据项Q已经被其他的事务加以非共享锁，则事务T必须等待，直到所有其他事务的非共享锁全部被释放。
  - 事务T可以释放它在任何数据项上所加的任何类型的锁。

锁的概念

- 银行数据库系统的例子：
  - 设A和B是两个帐号。事务T_7从帐号B向帐号A转50元钱，事务T_8显示帐号A和B的总金额。

T_7:	T_8:
LOCK-X(B);	LOCK-S(A);
READ(B);	READ(A);
B := B - 50;	UNLOCK(A);
WRITE(B);	LOCK-S(B);
UNLOCK(B);	READ(B);
LOCK-X(A);	UNLOCK(B);
READ(A);	DISPLAY(A+B)。
A := A + 50;	
WRITE(A);	
UNLOCK(A)。	

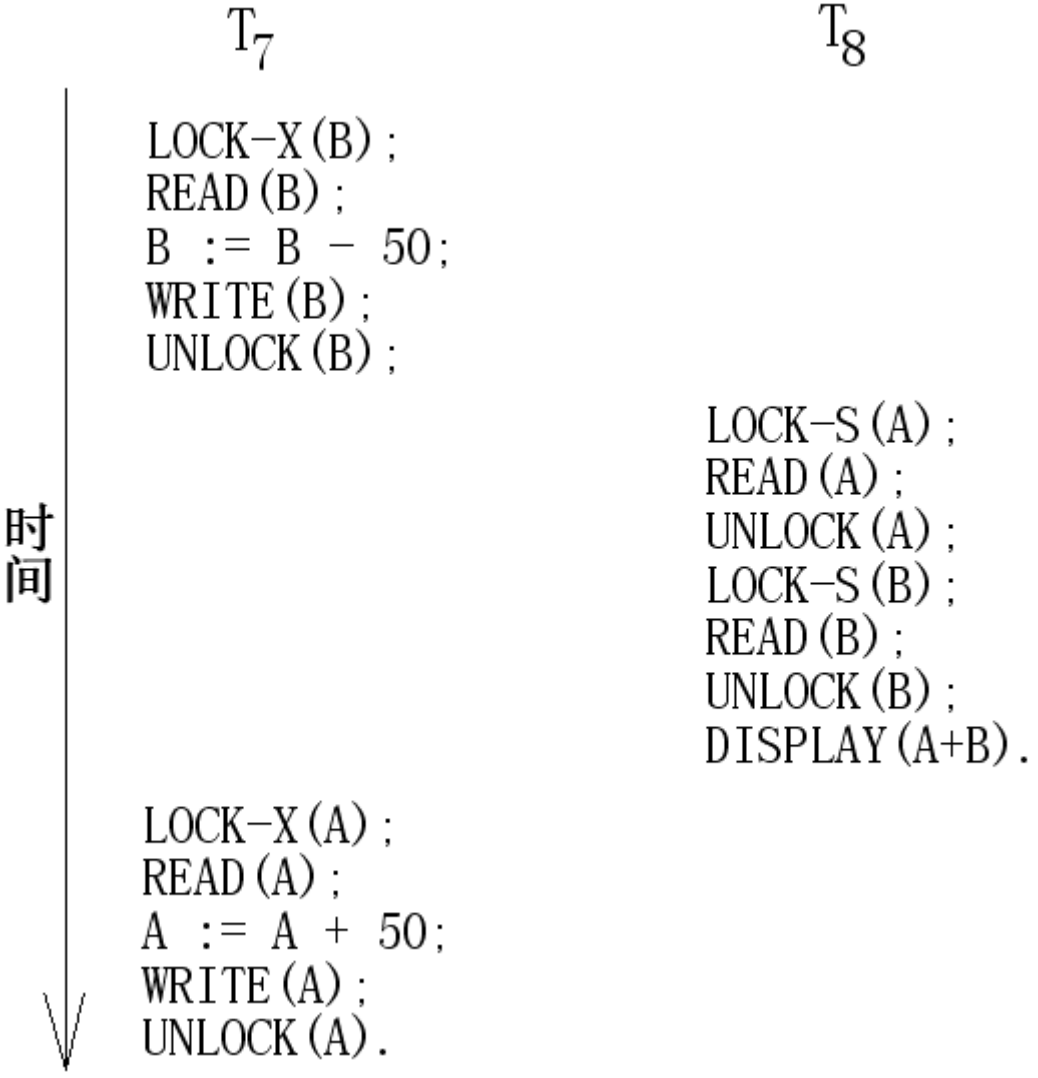
锁的概念

- 银行数据库系统的例子：
  - 设A和B的值分别是100和200元。
  - 调度1：
    - 两个事务串行执行，即<T_7, T_8>或<T_8, T_7>方式执行，T_8总是显示300美元的值。

```
•T7: LOCK-X(B);
•    READ(B);
•    B := B - 50;
•    WRITE(B);
•    UNLOCK(B);
•    LOCK-X(A);
•    READ(A);
•    A := A + 50;
•    WRITE(A);
•    UNLOCK(A)。
•T8: LOCK-S(A);
•    READ(A);
•    UNLOCK(A);
•    LOCK-S(B);
•    READ(B);
•    UNLOCK(B);
•    DISPLAY(A+B)
```

锁的概念

- 银行数据库系统的例子：
  - 调度2：
    - 事务T_8错误地显示250元。
    - 出现这种情况的原因是T_8所看到的是一个不一致的数据库状态。



锁的概念

- 由上例可见：
  - 一个事务只要存取一个数据项，它就必须持有该数据项上的一个锁。
  - 如果一个事务完成了对一个数据项的最后一次存取之后就立即放弃它的锁，则不能确保调度的可串行性。

锁的概念

- 修改事务T_7，把所有的释放锁操作放到最后，得到事务T_9：
  - LOCK-X(B);
  - READ(B);
  - B := B - 50;
  - WRITE(B);
  - LOCK-X(A);
  - READ(A);
  - A := A + 50;
  - WRITE(A);
  - UNLOCK(B);
  - UNLOCK(A)。
- 类似地，修改事务T_8得到事务T_{10}：
  - LOCK-S(A);
  - READ(A);
  - LOCK-S(B);
  - READ(B);
  - DISPLY(A+B);
  - UNLOCK(A);
  - UNLOCK(B)。

锁的概念

- T_9和T_{10}解决了T_7和T_8存在的问题。
- 但是，T_9和T_{10}存在死琐问题。

时间	T_9	T_{10}
↓	LOCK-X(B);	
	READ(B);	
	B := B - 50;	
	WRITE(B);	
		LOCK-S(A);
		READ(A);
		LOCK-S(B);
	LOCK-X(A);	

- 当死锁发生时，系统必须放弃至少一个处于死锁状态的事务，释放这个事务所加锁的数据项，使其他事务可以继续运行。

基于锁的并发控制协议

- 锁的概念
- **死锁、死锁的处理**
- 两段锁并发控制协议

死锁的处理

两类方法

1. 死锁预防

2. 死锁的检测与恢复

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

预防死锁的方法

- 一次封锁法
- 顺序封锁法

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
- 存在的问题
  - 降低系统并发度
  - 难于事先精确确定封锁对象

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

死锁的检测与恢复

- 死锁的诊断
  - ✓ 超时法
  - ✓ 事务等待图法

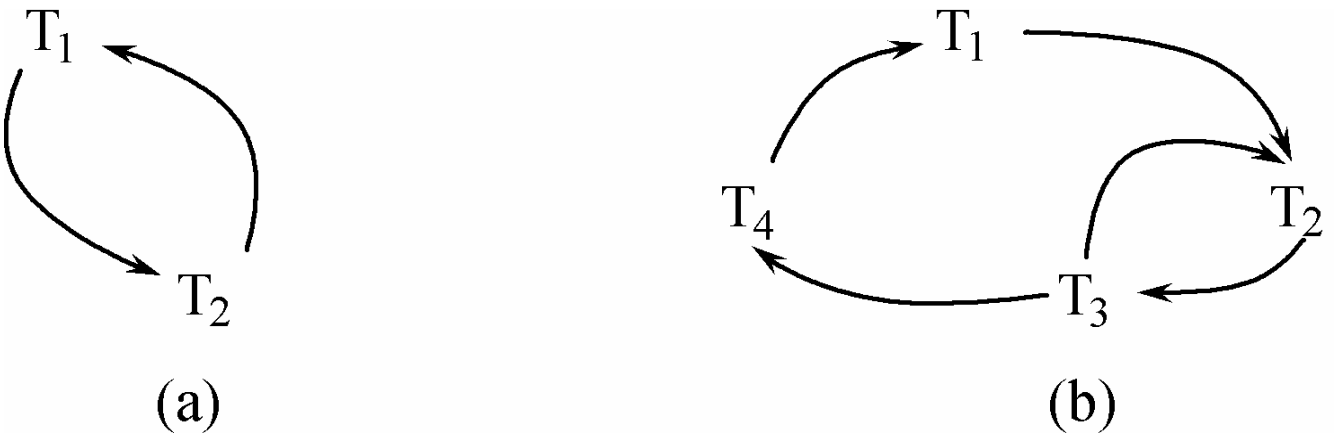
超时法

- 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

等待图法

- 用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



事务等待图

- 图(a)中，事务T1等待T2，T2等待T1，产生了死锁
- 图(b)中，事务T1等待T2，T2等待T3，T3等待T4，T4又等待T1，产生了死锁
- 图(b)中，事务T3可能还等待T2，在大回路中又有小的回路

等待图法

- 并发控制子系统周期性地(比如每隔数秒)生成等待图。如果发现图中存在回路，则表示系统中出现了死锁。

死锁的恢复

- 死锁的恢复
  - 选择牺牲者：必须决定回滚哪一个(或哪一些)事务以打破死锁，应使事务回滚代价最小
    - 事务已计算了多久，还将计算多长时间
    - 该事务已使用了多少数据项
    - 为完成事务还需使用多少数据项
    - 回滚时将牵涉多少事务
  - 回滚：彻底回滚、部分回滚
  - 饿死
    - 有可能同一事务总是被选为牺牲者，发生饿死
    - 因而必须保证一个事务被选为牺牲者的次数有限

基于锁的并发控制协议

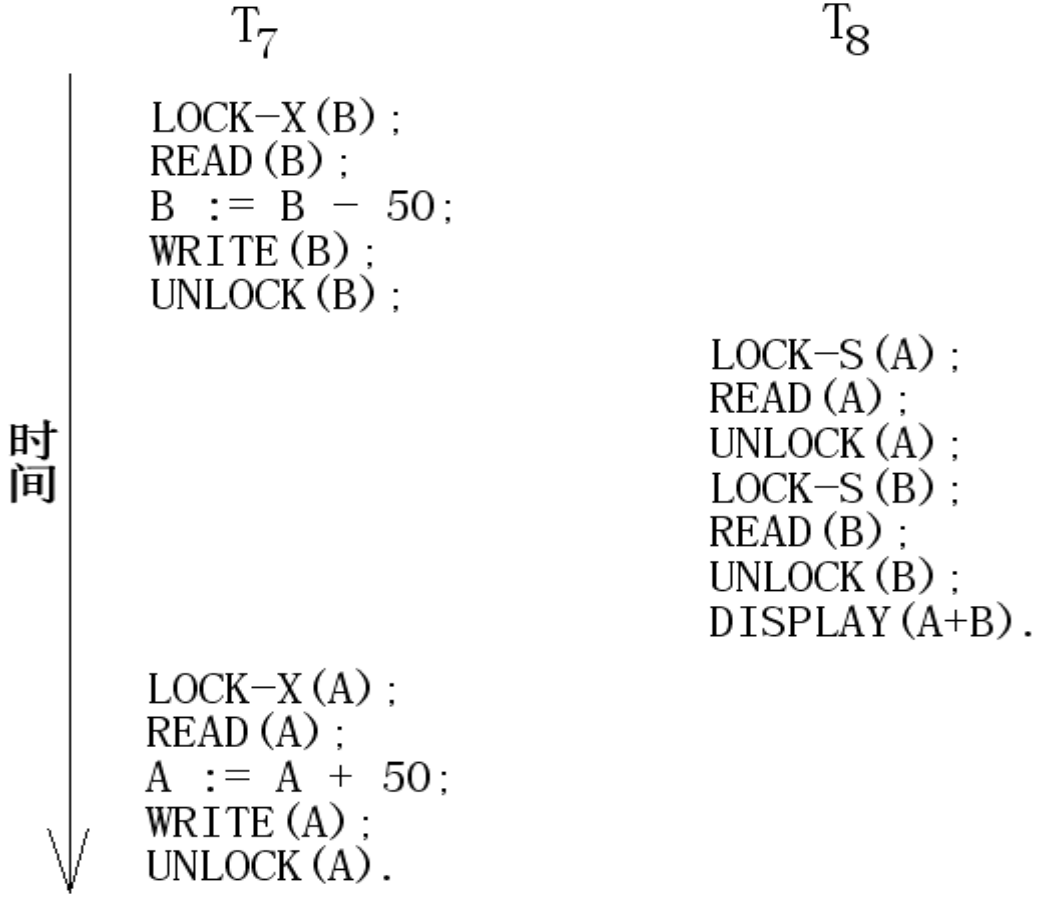
- 锁的概念
- 死锁、死锁的处理
- 两段锁并发控制协议

两段锁并发控制协议

- 两段锁协议
  - 两阶段锁协议要求每个事务分两个阶段进行数据项的加锁和解锁。
    - 阶段1 加锁阶段。在这阶段，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
    - 阶段2 解锁阶段。在这阶段，事务可以释放任何数据项上的任何类型的项，但是不能再申请任何项。
  - 每个事务开始运行后即进入加锁阶段，申请获得所需要的所有锁。
  - 当一个事务第一次释放锁时，该事务进入解锁阶段。进入解锁阶段的事务不能再申请任何锁。

两段锁并发控制协议

- 满足两段锁协议？ –No



两段锁并发控制协议

- 考虑如下两个事务T_{11}和T_{12}：
  - T_{11} T_{12}
  - READ(a_1); READ(a_1);
  - READ(a_2); READ(a_2);
  - ... DISPLAY(a_1+a_2)。
  - READ(a_n);
  - WRITE(a_1)。

  - 如果使用两段锁协议，T_{11}必须对a_1加互斥锁。于是，T_{11}和T_{12}的任何一种并发运行都实际上是串行运行。

  - 然而，T_{11}仅仅在最后执行WRITE(a_1)操作时才需要a_1上的互斥锁。如果T_{11}初始地对a_1加以共享锁，当执行WRITE(a_1)操作时再改为互斥锁，将会获得更高的并发性，T_{11}和T_{12}可以并发地存取a_1和a_2。

两段锁并发控制协议

- 从上例可以看到，可以进一步改善两段锁协议。
  - 用UPGRADE表示共享锁到互斥锁转换的操作，
  - 用DOWNGRADE表示互斥锁到共享锁转换操作。
  - 锁转换操作不可以随便乱用。UPGRADE只能在加锁阶段使用，DOWNGRADE只能在解锁阶段使用。
  - 事务T_{11}和T_{12}可以在改善的两段锁协议下并发运行。

T_{11}	T_{12}
LOCK-S(a_1)	
	LOCK-S(a_1)
LOCK-S(a_2)	
	LOCK-S(a_2)
LOCK-S(a_3)	
LOCK-S(a_4)	
	ULOCK(a_1)
	ULOCK(a_2)
LOCK-S(a_n)	
UPGRADE(a_1)	

两段锁并发控制协议

- 改善的两段锁协议：
  - 当事务T_i提交一个READ(Q)操作时，系统先执行LOCK-S(Q)操作，然后再执行READ(Q)操作。
  - 当事务T_i提交一个WRITE(Q)操作时，系统检查T_i是否已持有Q上的一个共享锁。如果是这样，系统先执行UPGRADE(Q)操作，然后再执行WRITE(Q)操作；否则，系统先执行LOCK-X(Q)操作，然后再执行WRITE(Q)操作。
- 可以证明，任何一个满足两段锁协议的合理调度都是可冲突可串行的。但是，并非每组具有冲突可串行调度的事务都有一个满足两段锁协议的调度。

并发控制协议

- 基于锁的协议
- **基于时间戳的协议**
- 多版本机制
- 快照隔离
- 等等….

基于时间戳的并发控制协议

- 另一种决定事务可串行化次序的方法是事先选定事务的次序，最常见的方法是**时间戳排序机制**
- 对于系统中每个事务T_i，我们把一个唯一的固定时间戳和它联系起来，记为$TS(T_i)$
  - 系统时钟
  - 逻辑计数器
- 如有一新事物T_j进入系统，则$TS(T_i) < TS(T_j)$

基于时间戳的并发控制协议

- 事务时间戳决定了串行化顺序，
  - 即若 $TS(T_i) < TS(T_j)$，则系统必须保证所产生的调度等价于事务T_i出现在事务T_j之前的某个串行调度

- 每个数据项Q需要与两个时间戳值关联
  - W-timestamp(Q):表示成功执行write(Q)的所有事务的最大时间戳
  - R-timestamp(Q):表示成功执行read(Q)的所有事务的最大时间戳

基于时间戳的并发控制协议

- 时间戳排序协议(1)
  - 假设事务T_i发出read(Q)
    - 若$TS(T_i)$<W-timestamp(Q)，则T_i需要读入的Q值已被覆盖。因此read操作被拒绝，T_i回滚
    - 若$TS(T_i)$>=W-timestamp(Q)，则执行read操作，R-timestamp(Q)被设置为max($TS(T_i)$, R-timestamp(Q))

基于时间戳的并发控制协议

- 时间戳排序协议(续)
  - 假设事务T_i发出write(Q)
    - 若$TS(T_i)$<R-timestamp(Q)，则T_i产生的Q值是先前所需要的值，且系统已假定该值不会产生。因此write操作被拒绝，T_i回滚
    - 若$TS(T_i)$<W-timestamp(Q)，则T_i试图写入的Q值已过时。因此write操作被拒绝，T_i回滚
    - 其他情况，则执行write操作，W-timestamp(Q)被设置为$TS(T_i)$

基于时间戳的并发控制协议

- 时间戳排序协议
  - 优点
    - 保证冲突可串行化
    - 保证无死锁
  - 缺点
    - 一系列冲突的短事务可能引起长事务反复重启，导致长事务饿死
    - 可能产生不可恢复的调度

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

图 15-17　调度 3

满足时间戳协议的调度

如果事务T_i失败了，除了需将事务T_i回滚外，同时必须确保那些依赖于T_i的任何事务T_j(即T_j读取了由T_i所写的数据)也必须同时回滚(级联回滚)

——可恢复的调度

基于时间戳的并发控制协议

- 假设TS(T_{27})<TS(T_{28})
  - T_{27}的read(Q)与T_{28}的write(Q)执行成功
  - 然而由于TS(T_{27})<W-timestamp(Q)=TS(T_{28}),则T_{27}需要回滚
  - 然而T_{27}回滚是没有必要的，因为其写入的值永远不会读到。

T_{27}	T_{28}
read(Q)	
	write(Q)
write(Q)	

图 15-18　调度 4

因而，Thomas写规则被提出

基于时间戳的并发控制协议

- Thomas写规则
  - 假设事务T_i发出read(Q)，处理方法与时间戳排序协议相同
  - 假设事务T_i发出write(Q)
    - 若$TS(T_i)$<R-timestamp(Q)，则T_i产生的Q值是先前所需要的值，且系统已假定该值不会产生。因此write操作被拒绝，T_i回滚
    - 若$TS(T_i)$<W-timestamp(Q)，则T_i试图写入的Q值已过时。因此这个write操作可忽略。
    - 其他情况，则执行write操作，W-timestamp(Q)被设置为$TS(T_i)$

并发控制协议

- 基于锁的协议
- 基于时间戳的协议
- 多版本机制
- 快照隔离
- 等等….

多版本机制

- 目前所讲述的并发控制机制
  - 要么延迟一项操作——基于锁的协议
  - 要么中止发出该操作的事务——基于时间戳的协议

来保证可串行性

- 出现的问题，例如
  - read操作可能由于相应值还未写入而延迟
  - read操作因为它要读的值被覆盖而被拒绝
- 解决方法
  - 如果每一数据项的旧值拷贝可以保存在系统中

——多版本的并发控制协议

多版本机制

- 多版本并发控制的主要思想
  - 每个write(Q)操作创建Q的一个新版本
  - 当事务发出一个read(Q)操作时，并发控制管理器选择Q的一个版本进行读取
  - 必须保证用于读取的版本的选择能保持可串行性
  - 一个事务能够容易而快速地判定读取哪个版本的数据项是关键
- 可分为多版本时间戳排序机制和多版本两段锁机制

多版本机制

- 多版本时间戳排序机制
  - 对于每个数据项Q，有一个版本序列$<Q_1,Q_2,\cdots Q_m>$与之关联，每个版本Q_k包含三个数据字段
    - Content是Q_k的版本值
    - W-timestamp(Q_k)是创建Q_k版本的事务的时间戳
    - R-timestamp(Q_k)是所有成功地读取Q_k版本的事务的最大时间戳
  - 事务T_i通过发出write(Q)操作来创建数据项Q的一个新版本Q_k，并将W-timestamp(Q_k)和R-timestamp(Q_k)初始化为TS(T_i)

多版本机制

- 多版本时间戳排序机制(续)
  - 设事务T_i发出read(Q)或write(Q)操作，Q_k满足W-timestamp(Q_k)=max{W-timestamp(Q_j)<=TS(T_i)}
  - (1)如果事务T_i发出的是read(Q)，则返回Q_k
    - 一个事务读取位于其前的最近版本
  - (2)如果事务T_i发出write(Q)，且TS(T_i)<R-timestamp(Q_k)，则系统回滚T_i
    - T_i试图写入其他事务应该已读取的版本，写操作太迟了，则中止

多版本机制

- 多版本时间戳排序机制(续)
  - (3)如果事务T_i发出write(Q),且TS(T_i)<W-timestamp(Q_k)，则系统覆盖Q_k；否则，若TS(T_i)>=R-timestamp(Q_k)，则创建Q的一个新版本
- 不再需要的版本删除
  - 如果某数据项的两个版本Q_k和Q_j，这两个版本的W-timestamp都小于系统中最老的事务的时间戳，那么删除Q_k和Q_j中较旧的版本

多版本机制

- 多版本时间戳排序机制(续)
  - 优点
    - 读请求从不失败且不必等待
  - 缺点
    - 读取数据项要求更新R-timestamp字段，产生两次潜在的磁盘访问
    - 事务冲突通过回滚而不是等待来解决，开销可能大
    - 不保证可恢复性和无级联性

多版本机制

- 多版本两段锁机制
  - 该协议对只读事务(read-only transaction)与更新事务(update transaction)加以区分
  - 数据项每个版本有一个时间戳ts-counter，可不基于时钟，仅为一个计数器

多版本机制

- 多版本两段锁机制(续)
  - 更新事务执行强两段锁协议，即它们持有全部锁直到事务结束。
  - 在执行前，数据库系统读取ts-counter当前值作为该事务的时间戳，
  - 更新事务Ti完成其任务后，按如下方式进行提交
    - 首先，Ti将它创建的每一版本的时间戳设置为ts-counter的值加1
    - Ti将ts-counter增加1
  - 在同一时间内只允许有一个更新事务进行提交

多版本机制

- 多版本两段锁机制(续)
  - 只读事务均不必等待加锁
  - 只读事务在执行前，数据库系统读取ts-counter当前值作为该事务的时间戳，在执行读操作是遵从多版本时间戳排序协议
- 版本删除类似于多版本时间戳排序协议
- 多版本两段锁保证了调度是可恢复的和无级联的

并发控制协议

- 基于锁的协议
- 基于时间戳的协议
- 多版本机制
- **快照隔离**
- 等等….

快照隔离

- 是一种特殊的并发控制机制，在商业和开源系统中广泛接受
  - Oracle、PostgreSQL、SQL Sever等
- 在事务开始执行时给它数据库的一份“快照”，和其他并发事务完全隔离开
  - 对于只读事务是理想的，不用等待、不会中断
  - 对于更新写，必须处理其他并发更新的事务之间的潜在冲突
    - 当允许事务T提交时，事务T变为提交状态，并且T对数据库的所有写操作都必须作为一个原子操作执行

总结

- 本章重点
  - 掌握事务的概念及性质
  - 掌握事务调度和并发控制的概念
  - 掌握基于锁的并发控制协议
  - 掌握死锁检测

Now let's go to

Next Chapter