

# 第11章：并发控制

## Concurrency Control

李东博

哈尔滨工业大学  
计算学部  
物联网与泛在智能研究中心  
电子邮件: [ldb@hit.edu.cn](mailto:ldb@hit.edu.cn)

2023年春

# 教学内容<sup>1</sup>

- 1 Transactions
- 2 Concurrency Control
  - Schedules
  - Isolation Levels
  - Serializability
- 3 Lock-based Concurrency Control
  - Locks
  - Two-Phase Locking (2PL)
  - Deadlocks
  - Multi-Granularity Locking
  - Phantoms
- 4 Timestamp Ordering (T/O) Concurrency Control
  - Basic T/O
  - Optimistic Concurrency Control (OCC)
- 5 Multi-Version Concurrency Control (MVCC)

---

<sup>1</sup>课件更新于2023年2月16日

# Transactions

# 事务(Transaction)

**事务**是在数据库上执行的一个或多个操作构成的序列，用来完成数据库系统的高级功能

- 事务的操作**要么全部执行，要么一个也不执行**

## Example (事务)

账户A给账户B转账100元

- ① 查询账户A的余额
- ② 如果余额大于等于100元，则执行步骤3和4；否则，中止
- ③ 从账户A中扣除100元
- ④ 在账户B中增加100元

# SQL事务语句(Transaction Statement)

事务启动(start): **BEGIN;**

事务提交(commit): **COMMIT;**

- 将事务对数据库的修改持久地写入数据库中

事务中止(abort): **ROLLBACK;**

- 将事务对数据库的修改全部撤销(undo), 就好像从未执行过该事务
- 事务既可以被自己中止, 也可以被DBMS中止

# 事务的ACID性质(The ACID Properties)

原子性(Atomicity): “All or nothing”

- 事务的操作要么全部执行，要么一个也不执行

一致性(Consistency): “It looks correct to me”

- 如果事务的程序正确，并且事务启动时数据库处于一致状态(consistent state)，则事务结束时数据库一定处于一致状态

隔离性(Isolation): “As if alone”

- 一个事务的执行不受其他事务的干扰

持久性(Durability): “Survive failures”

- 事务一旦提交，则它对数据库的修改一定全部持久地写到数据库中

# 原子性(Atomicity)—“All or Nothing”

事务的执行只有两种结局

- 执行完所有操作后提交(commit)⇒不破坏原子性☺
- 执行了部分操作后中止(abort)⇒破坏原子性☹

事务的原子性由DBMS保证

- 已中止事务(aborted txn)执行过的操作必须撤销(undo)

## 第12章：故障恢复(Crash Recovery)

# 持久性(Durability)—“Survive Failures”

故障(failure)导致事务对数据库的修改有4种结果

- 已提交事务的修改已全部持久存储 $\implies$ 不破坏持久性☺
- 已提交事务的修改仅部分持久存储 $\implies$ 破坏持久性☹
- 已中止事务的修改有些已持久存储 $\implies$ 破坏持久性☹
- 已中止事务的修改未持久存储 $\implies$ 不破坏持久性☺

事务的持久性由DBMS保证

- 重做(redo)已提交事务对数据库的修改
- 撤销(undo)已中止事务对数据库的修改

## 第12章：故障恢复(Failure Recovery)



# 一致性(Consistency)—“It Looks Correct to Me”

事务的一致性由用户保证

- 别写错程序

# 隔离性(Isolation)—“As If Alone”

多个事务的执行有两种方式

- 串行执行(serial execution)  $\implies$  不破坏隔离性 😊
- 交叉执行(interleaving execution)  $\implies$  可能破坏隔离性 😞

事务的隔离性由DBMS保证

- 并发控制(concurrency control): 确定多个事务的操作的正确交叉执行顺序

## 第11章：并发控制(Concurrency Control)

# Concurrency Control

# Concurrency Control Schedules

# 基本概念

**数据库(database)**: 固定的数据对象集合(a fixed set of data objects)

- **数据对象**是个抽象概念, 可以是属性值、元组、页、关系或数据库
- 数据对象又称**数据库元素(database element)**
- 我们先不考虑数据的插入和删除

**事务(transaction)**: 数据库对象的读/写操作序列

- 事务可以在数据库上进行很多操作, 但DBMS只关心事务对数据对象的读/写操作

# 调度(Schedule)

调度是一个或多个事务的重要操作(action)的序列

## Example (调度)

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

## 串行调度 (Serial Schedule)

如果一个调度中不同事务的操作没有交叉, 则该调度是串行调度

### Example (串行调度)

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

# 非串行调度(Nonserial Schedule)

不是串行调度的调度称为非串行调度

## Example (非串行调度)

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



## 调度的正确性(Correctness)

单独执行每个事务都会将数据库从一种一致状态(consistent state)变为另一种一致状态

### Example (调度的正确性)

设数据库的一致性约束条件(consistency constraint)为  $A = B$

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

# 串行调度的正确性

任意串行调度都能保持数据库的一致性

## Example (串行调度的正确性)

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

## 串行调度的正确性(续)

不同的串行调度可能导致数据库处于不同的最终状态，但都是一致状态

### Example (串行调度的正确性)

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

# 不正确的非串行调度

非串行调度可能会破坏数据库的一致性

## Example (不正确的非串行调度)

$T_1$	$T_2$	$A$	$B$
		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

# 异常(Anomaly)

非串行调度会导致事务的异常行为(anomaly behavior)，从而破坏数据库的一致性

- 脏写(dirty write)
- 脏读(dirty read)
- 不可重复读(unrepeatable read)
- 幻读(phantom)

## 脏写 (Dirty Write)

在事务  $T_2$  提交前,  $T_2$  写的  $A$  值被事务  $T_1$  修改

### Example (脏写)

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

## 脏读(Dirty Read)

在事务 $T_1$ 提交前， $T_1$ 写的 $A$ 值被事务 $T_2$ 读取

### Example (脏读)

$T_1$	$T_2$	$A$	$B$
		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

## 不可重复读(Unrepeatable Read)

- 事务  $T_2$  修改了事务  $T_1$  读过的对象  $A$  的值, 且  $T_2$  提交
- 如果  $T_1$  再次读  $A$  的值, 那么它将得到一个不同于前一次读到的  $A$  值, 即便  $T_1$  在此期间并未对  $A$  进行修改

### Example (不可重复读)

设一致性约束条件为  $A \geq 0$

$T_1$	$T_2$	$A$
		1
READ(A, t)		
	READ(A, s)	
	s := s - 1	
	WRITE(A, s)	0
	COMMIT	
READ(A, t)		
t := t - 1		
WRITE(A, t)		-1



# 幻读(Phantom)

涉及数据的插入和删除，后面再讲

## 等价调度(Equivalent Schedule)

如果两个调度在任意数据库实例上的效果都相同，则这两个调度等价

### Example (等价调度)

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

# 可串行化调度(Serializable Schedule)

如果一个调度等价于一个串行调度，则该调度是**可串行化调度**

## Example (可串行化调度)

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

## 可串行化调度的优点

	脏写	脏读	不可重复读	幻读
可串行化	无	无	无	无

并发事务全部由DBMS来调度，程序员无需考虑事务并发的问题

# 可串行化调度的缺点

可串行化调度的并发度低

在某些应用场景下，并发事务不需要严格隔离

- 一个事务对部分对象的修改可以暴露给对其他并发事务
- 弱隔离级别(*weaker isolation level*)可以提高系统的并发度

# Concurrency Control

## Isolation Levels

# 隔离级别(Isolation Level)

SQL-92规定了并发事务的4种隔离级别

- 读未提交(Read Uncommitted)
- 读提交(Read Committed)
- 可重复读(Repeatable Read)
- 可串行化(Serializable)

在不同隔离级别下，一个事务修改过的对象的值对其他并发事务的可见程度不同

## 设置事务的隔离级别

- PostgreSQL: `BEGIN TRANSACTION ISOLATION LEVEL {  
SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ  
UNCOMMITTED };`
- MySQL: `SET TRANSACTION ISOLATION LEVEL { SERIALIZABLE |  
REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED };`

# Read Uncommitted

未提交事务所做的修改对其他事务可见

## Example (Read Uncommitted)

$T_1$	$T_2$	A	Variables
		A = 1	
READ(A, t)			t = 1
	READ(A, s)		s = 1
	s := s * 2		s = 2
	WRITE(A, s)	A = 2	
READ(A, x)			x = 2
	COMMIT		
READ(A, y)			y = 2
COMMIT			
READ(A, z)			z = 2



# Read Uncommitted下的异常

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	可能	可能	可能

# Read Committed

只有已提交的事务所做的修改才对其他事务可见 ▶ 演示

## Example (Read Committed)

$T_1$	$T_2$	A	Variables
		A = 1	
READ(A, t)			t = 1
	READ(A, s)		s = 1
	s := s * 2		s = 2
	WRITE(A, s)	A = 2	
READ(A, x)			x = 1
	COMMIT		
READ(A, y)			y = 2
COMMIT			
READ(A, z)			z = 2

PostgreSQL的默认隔离级别是Read Committed

# Read Committed下的异常

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	可能	可能	可能
READ COMMITTED	无	可能	可能

## 可重复读(Repeatable Read)

如果一个事务不修改对象X的值，则该事务在任何时候读到的X值都等于事务启动时读到的X值 ▶ 演示

### Example (Repeatable Read)

$T_1$	$T_2$	A	Variables
		A = 1	
READ(A, t)			t = 1
	READ(A, s)		s = 1
	s := s * 2		s = 2
	WRITE(A, s)	A = 2	
READ(A, x)			x = 1
	COMMIT		
READ(A, y)			y = 1
COMMIT			
READ(A, z)			z = 2

MySQL的默认隔离级别是Repeatable Read

# Repeatable Read下的异常

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	可能	可能	可能
READ COMMITTED	无	可能	可能
REPEATABLE READ	无	无	可能

## 可串行化(Serializable)

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	可能	可能	可能
READ COMMITTED	无	可能	可能
REPEATABLE READ	无	无	可能
SERIALIZABLE	无	无	无

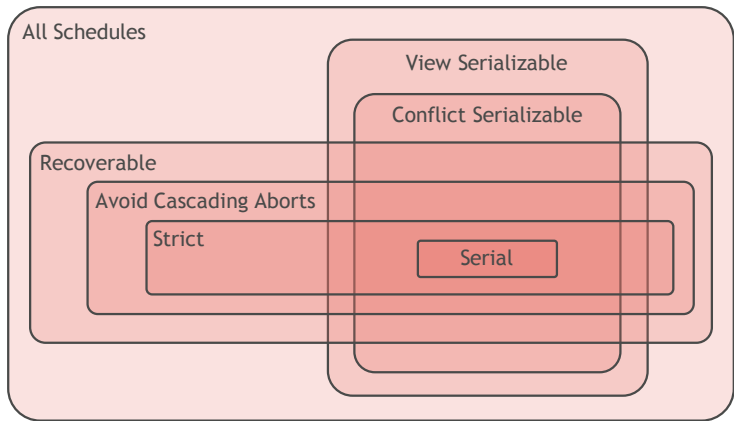
# Concurrency Control

## Serializability

# 冲突可串行化(Conflict Serializability)

支持可串行化隔离级别的DBMS实施(enforce)的都是冲突可串行化

- 冲突可串行化比一般可串行化的条件更严
- 冲突可串行化更便于在DBMS中实施





# 冲突(Conflict)

如果两个操作满足以下三个条件，则这两个操作**冲突**

- 这两个操作属于**不同的事务**
- 这两个操作涉及**相同的对象**
- 这两个操作中**至少有一个操作是写**

## 写-写冲突(Write-Write Conflict/W-W Conflict)

写-写冲突可能导致脏写(dirty write)

### Example (写-写冲突)

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

# 写-读冲突(Write-Read Conflict/W-R Conflict)

写-读冲突可能导致脏读(dirty read)

## Example (写-读冲突)

$T_1$	$T_2$	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

# 读-写冲突(Read-Write Conflicts/R-W Conflict)

读-写冲突可能导致不可重复读(unrepeatable read)

## Example (读-写冲突)

$T_1$	$T_2$	$A$
		1
READ(A, t)		
	READ(A, s)	
	s := s - 1	
	WRITE(A, s)	0
	COMMIT	
READ(A, t)		
t := t - 1		
WRITE(A, t)		-1

# 事务的简化表示

对DBMS而言，事务中与调度相关的操作只有对象的读和写

## Example (事务的简化表示)

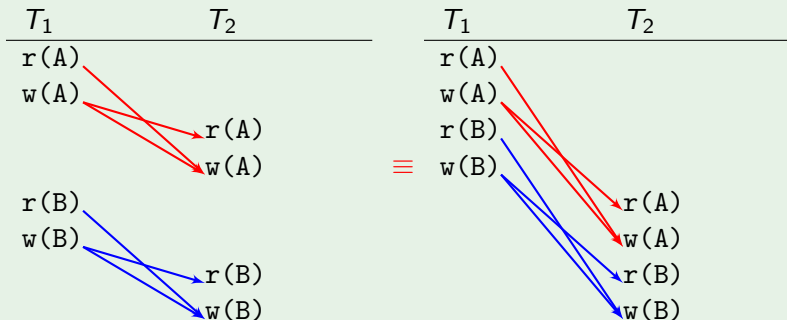
$T_1$	$T_2$		$T_1$	$T_2$
READ(A, t)			r(A)	
t := t + 100				
WRITE(A, t)			w(A)	
	READ(A, s)			r(A)
	s := s * 2			
	WRITE(A, s)	→		w(A)
READ(B, t)			r(B)	
t := t + 100				
WRITE(B, t)			w(B)	
	READ(B, s)			r(B)
	s := s * 2			
	WRITE(B, s)			w(B)

# 冲突等价(Conflict Equivalence)

两个调度冲突等价(conflict equivalent)，如果

- 这两个调度涉及相同事务的相同操作
- 每一对冲突的操作在两个调度中的顺序都相同

## Example (冲突等价)



# 冲突可串行化调度 (Conflict Serializable Schedule)

如果一个调度冲突等价于一个串行调度, 则该调度是冲突可串行化调度

## Example (冲突可串行化调度)

$T_1$	$T_2$		$T_1$	$T_2$
r(A)			r(A)	
w(A)			w(A)	
	r(A)		r(B)	
	w(A)	$\equiv$	w(B)	
r(B)				r(A)
w(B)				w(A)
	r(B)			r(B)
	w(B)			w(B)

## 冲突可串行化调度(续)

如果通过将调度 $S$ 中不同事务中的非冲突操作交换顺序可以将 $S$ 转换为一个串行调度, 则 $S$ 是一个冲突可串行化调度

### Example (冲突可串行化调度)

$T_1$	$T_2$		$T_1$	$T_2$		$T_1$	$T_2$		$T_1$	$T_2$
r(A)			r(A)			r(A)			r(A)	
w(A)			w(A)			w(A)			w(A)	
	r(A)			r(A)		r(B)			r(B)	
	w(A)	$\equiv$		r(B)	$\equiv$		r(A)	$\equiv$		r(A)
r(B)				w(A)			w(A)			w(B)
w(B)			w(B)			w(B)				w(A)
	r(B)			r(B)			r(B)			r(B)
	w(B)			w(B)			w(B)			w(B)



# 非冲突可串行化调度

不是冲突可串行化的调度称作非冲突可串行化调度

## Example (非冲突可串行化调度)

$T_1$	$T_2$		$T_1$	$T_2$		$T_1$	$T_2$
r(A)			r(A)				r(A)
	r(A)		w(A)				w(A)
	w(A)		r(B)				r(B)
w(A)		$\neq$	w(B)		$\neq$		w(B)
r(B)				r(A)		r(A)	
w(B)				w(A)		w(A)	
	r(B)			r(B)		r(B)	
	w(B)			w(B)		w(B)	

# 冲突可串行化测试(Conflict Serializability Test)

**第1步:** 将调度 $S$ 表示为**优先图**(precedence graph)

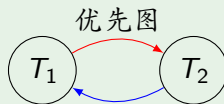
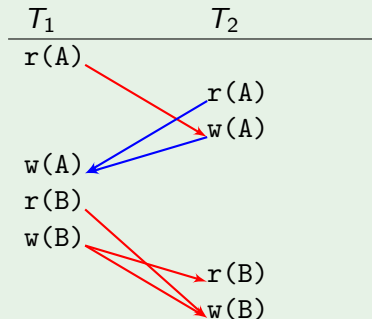
- 每个顶点代表 $S$ 中的一个事务
- 从事务 $T_i$ 到事务 $T_j$ 有一条有向边(arc), 如果 $T_i$ 的某个操作 $o_i$ 和 $T_j$ 的某个操作 $o_j$ 冲突, 并且 $o_i$ 在 $S$ 中先于 $o_j$

**第2步:**  $S$ 是**冲突可串行化调度**, 当且仅当其优先图没有环(acyclic)

- 优先图顶点的任意拓扑序(topological order)表示了一个与 $S$ 冲突等价的串行调度

# 冲突可串行化测试示例1

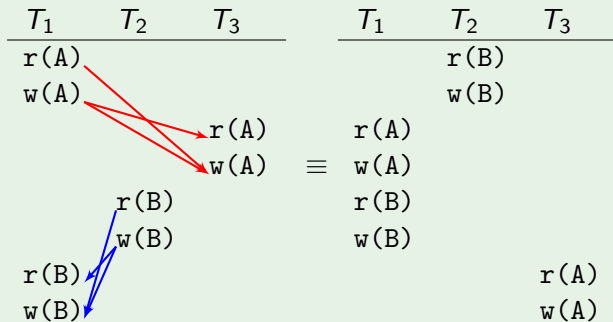
## Example (冲突可串行化测试)



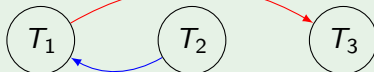
该调度不是冲突可串行化调度

## 冲突可串行化测试示例2

### Example (冲突可串行化测试)



优先图

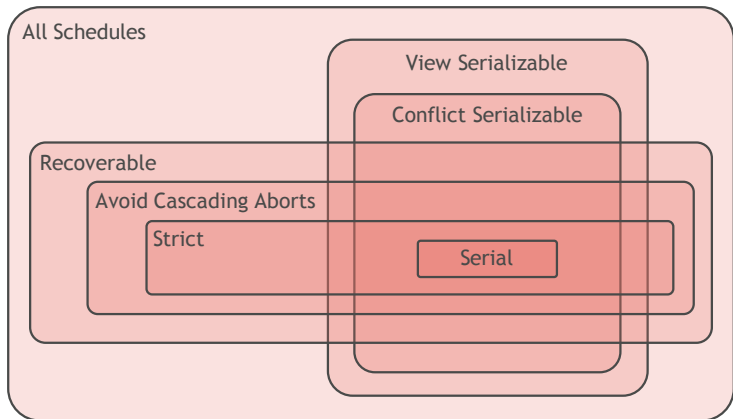


该调度是冲突可串行化调度

# 视图可串行化(View Serializability)

视图可串行化是比冲突可串行化更弱的概念

- 测试和实施视图可串行化很难
- 没有DBMS实施视图可串行化



# 视图可串行化调度(View Serializable Schedule)

两个调度 $S_1$ 和 $S_2$ 视图等价(view equivalent), 如果

- 如果 $S_1$ 中事务 $T_1$ 读了对象 $A$ 的初始值, 则 $S_2$ 中 $T_1$ 也读了 $A$ 的初始值
- 如果 $S_1$ 中事务 $T_1$ 读了事务 $T_2$ 修改过的 $A$ 值, 则 $S_1$ 中 $T_1$ 也读了 $T_2$ 修改过的 $A$ 值
- 如果 $S_1$ 中事务 $T_1$ 写了 $A$ 的最终值, 则 $S_2$ 中 $T_1$ 也写了 $A$ 的最终值

如果一个调度视图等价于一个串行调度, 则该调度是视图可串行化调度

## Example (视图可串行化调度)

$T_1$	$T_2$	$T_3$		$T_1$	$T_2$	$T_3$
r(A)				r(A)		
	w(A)		$\equiv_{view}$	w(A)		
w(A)					w(A)	
		w(A)				w(A)

# 冲突可串行化vs视图可串行化

冲突可串行化调度一定是视图可串行化调度

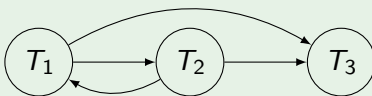
视图可串行化调度不一定是冲突可串行化调度

- 视图可串行化但非冲突可串行化的调度一定包含盲写(blind write)

## Example

$T_1$	$T_2$	$T_3$		$T_1$	$T_2$	$T_3$
r(A)				r(A)		
	w(A)		$\equiv_{view}$	w(A)		
w(A)					w(A)	
		w(A)				w(A)

优先图



该调度是视图可串行化调度，但不是冲突可串行化调度

# 并发控制协议(Concurrency Control Protocol)

并发控制协议用于对并发事务实施正确的(运行时)调度，而无需预先确定整个(静态)调度

悲观并发控制(pessimistic concurrency control)

- 假定冲突很多
- 不允许任何冲突发生

乐观并发控制(optimistic concurrency control)

- 假定冲突很少
- 冲突发生了，再去解决



# 并发控制协议的分类

方法1: 基于锁的并发控制(lock-based concurrency control)

方法2: 时间戳定序的并发控制(timestamp ordering concurrency control)

方法3: 多版本并发控制(multi-version concurrency control, MVCC)

# Lock-based Concurrency Control

# Lock-based Concurrency Control

## Locks

# 锁(Lock)

用锁来保护数据库对象

- 事务  $T_i$  只有获得了对象  $A$  的锁，才能读或写  $A$
- 如果事务  $T_i$  请求了对象  $A$  的锁，但并未获得，则  $T_i$  开始等待，直至获得  $A$  的锁为止
- 如果事务  $T_i$  已经获得了对象  $A$  的锁，则在  $T_i$  完成对  $A$  的操作后， $T_i$  必须释放  $A$  的锁

# 使用锁的并发事务执行

- LOCK(A): 请求对A加锁
- UNLOCK(A): 释放A的锁

## Example (锁的使用)

$T_1$	$T_2$	调度器的动作
LOCK(A)		将A上的锁授予 $T_1$
r(A)		
	LOCK(A)	
w(A)		
r(A)		
UNLOCK(A)		
	r(A)	
	w(A)	
	UNLOCK(A)	

# 使用锁的并发事务执行

- LOCK(A): 请求对A加锁
- UNLOCK(A): 释放A的锁

## Example (锁的使用)

$T_1$	$T_2$	调度器的动作
LOCK(A)		将A上的锁授予 $T_1$
r(A)		
	LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		
	r(A)	
	w(A)	
	UNLOCK(A)	

# 使用锁的并发事务执行

- LOCK(A): 请求对A加锁
- UNLOCK(A): 释放A的锁

## Example (锁的使用)

$T_1$	$T_2$	调度器的动作
LOCK(A)		将A上的锁授予 $T_1$
r(A)		
	LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	
	w(A)	
	UNLOCK(A)	

# 使用锁的并发事务执行

- LOCK(A): 请求对A加锁
- UNLOCK(A): 释放A的锁

## Example (锁的使用)

$T_1$	$T_2$	调度器的动作
LOCK(A)		将A上的锁授予 $T_1$
r(A)		
	LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	将A上的锁授予 $T_2$
	w(A)	
	UNLOCK(A)	



# 使用锁的并发事务执行

- LOCK(A): 请求对A加锁
- UNLOCK(A): 释放A的锁

## Example (锁的使用)

$T_1$	$T_2$	调度器的动作
LOCK(A)		将A上的锁授予 $T_1$
r(A)		
	LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	将A上的锁授予 $T_2$
	w(A)	
	UNLOCK(A)	释放A上的锁

# 锁的类型

## 共享锁(shared lock)/S锁(S-lock)

- 事务  $T_i$  只有获得了对象  $A$  的共享锁，才能读  $A$

## 互斥锁(exclusive lock)/X锁(X-lock)

- 事务  $T_i$  只有获得了对象  $A$  的互斥锁，才能写  $A$
- $T_i$  获得了  $A$  的互斥锁后，亦可读  $A$

## 锁的相容性(Compatibility)

如果对象A上有事务 $T_i$ 加的共享锁，则事务 $T_j$ 还可以对A加共享锁，但不可以对A加互斥锁

- 否则产生读-写冲突

如果对象A上有事务 $T_i$ 加的互斥锁，则事务 $T_j$ 对A既不能加共享锁，也不能加互斥锁

- 否则产生写-读冲突或写-写冲突

### 相容矩阵(compatibility matrix)

		请求加锁的类型	
		共享锁	互斥锁
已加锁 的类型	共享锁	Yes	No
	互斥锁	No	No

## 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

### Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	
w(A)		
UNLOCK(A)		
	r(A)	
	w(A)	
S-LOCK(A)		
	UNLOCK(A)	
r(A)		
UNLOCK(A)		

# 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

## Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		
	r(A)	
	w(A)	
S-LOCK(A)		
	UNLOCK(A)	
r(A)		
UNLOCK(A)		

## 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

### Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	
	w(A)	
S-LOCK(A)		
	UNLOCK(A)	
r(A)		
UNLOCK(A)		

# 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

## Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	A上的互斥锁授予 $T_2$
	w(A)	
S-LOCK(A)		
	UNLOCK(A)	
r(A)		
UNLOCK(A)		

## 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

### Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	A上的互斥锁授予 $T_2$
	w(A)	
S-LOCK(A)		拒绝
	UNLOCK(A)	
r(A)		
UNLOCK(A)		



## 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

### Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	A上的互斥锁授予 $T_2$
	w(A)	
S-LOCK(A)		拒绝
	UNLOCK(A)	释放A上的互斥锁
r(A)		
UNLOCK(A)		

## 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

### Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	A上的互斥锁授予 $T_2$
	w(A)	
S-LOCK(A)		拒绝
	UNLOCK(A)	释放A上的互斥锁
r(A)		A上的共享锁授予 $T_1$
UNLOCK(A)		

# 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

## Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	A上的互斥锁授予 $T_2$
	w(A)	
S-LOCK(A)		拒绝
	UNLOCK(A)	释放A上的互斥锁
r(A)		A上的共享锁授予 $T_1$
UNLOCK(A)		释放A上的共享锁

# 使用共享锁和互斥锁的并发事务执行

- S-LOCK(A): 请求对A加共享锁
- X-LOCK(A): 请求对A加互斥锁

## Example (共享锁和互斥锁的使用)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
UNLOCK(A)		释放A上的互斥锁
	r(A)	A上的互斥锁授予 $T_2$
	w(A)	
S-LOCK(A)		拒绝
	UNLOCK(A)	释放A上的互斥锁
r(A)		A上的共享锁授予 $T_1$
UNLOCK(A)		释放A上的共享锁

尽管使用锁来保护数据对象，但该调度不是冲突可串行化调度

# Lock-based Concurrency Control

## Two-Phase Locking (2PL)

# 两阶段锁协议(Two-Phase Locking, 2PL)

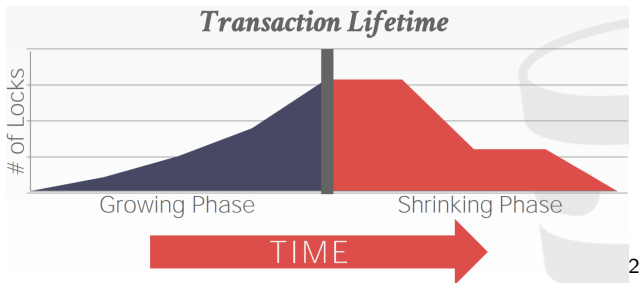
每个事务的执行分为两个阶段

## 增长阶段(Growing Phase)

- 事务向锁管理器(lock manager)请求需要的锁

## 萎缩阶段(Shrinking Phase)

- 事务释放它获得的锁，但不能再请求加锁



<sup>2</sup>来源: Andy Pavlo, CMU 15-445

# 基于2PL的并发事务执行

## Example (2PL)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	
w(A)		
r(A)		
UNLOCK(A)		
	r(A)	
	w(A)	
	UNLOCK(A)	

# 基于2PL的并发事务执行

## Example (2PL)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		
	r(A)	
	w(A)	
	UNLOCK(A)	



# 基于2PL的并发事务执行

## Example (2PL)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	
	w(A)	
	UNLOCK(A)	

# 基于2PL的并发事务执行

## Example (2PL)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	将A上的互斥锁授予 $T_2$
	w(A)	
	UNLOCK(A)	

# 基于2PL的并发事务执行

## Example (2PL)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	将A上的互斥锁授予 $T_2$
	w(A)	
	UNLOCK(A)	释放A上的互斥锁

# 基于2PL的并发事务执行

## Example (2PL)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	X-LOCK(A)	拒绝
w(A)		
r(A)		
UNLOCK(A)		释放A上的锁
	r(A)	将A上的互斥锁授予 $T_2$
	w(A)	
	UNLOCK(A)	释放A上的互斥锁

该调度是冲突可串行化调度(如何证明?)

## 2PL的优缺点

优点: 2PL能够保证冲突可串行化

缺点1: 2PL面临着级联中止(cascading abort)的问题

缺点2: 2PL可能导致死锁(deadlock)

## 级联中止(Cascading Abort)

一个事务中止可能会导致其他事务级联中止

### Example (级联中止)

$T_1$	$T_2$
X-LOCK(A)	
X-LOCK(B)	
r(A)	X-LOCK(A)
w(A)	
UNLOCK(A)	
	r(A)
	w(A)
r(B)	
w(B)	
ABORT	
	r(A)

- 2PL允许该调度
- $T_1$ 中止时,  $T_2$ 也要中止(级联中止), 否则 $T_1$ 修改过的信息将通过 $T_2$ 泄露给外界

# 严格调度(Strict Schedule)

如果一个调度中任意事务在结束前由它写入的数据库对象的值没有被其他事务读过或修改过，则该调度被称为严格调度

- 严格调度不会引发级联中止

# 强两阶段锁(Strong Strict Two-Phase Locking, SS2PL/Rigorous 2PL)

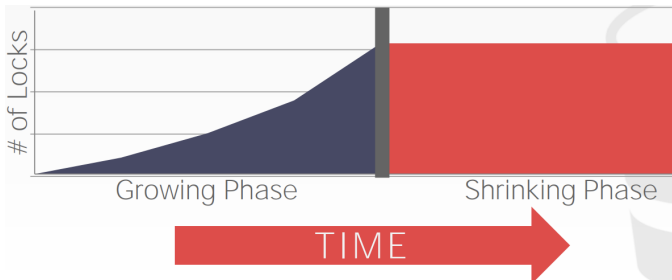
## 增长阶段(Growing Phase)

- 与2PL的增长阶段相同

## 萎缩阶段(Shrinking Phase)

- 当事务结束时，释放它获得的全部的锁

SS2PL保证生成严格的冲突可串行化调度，因而不会产生级联中止



3

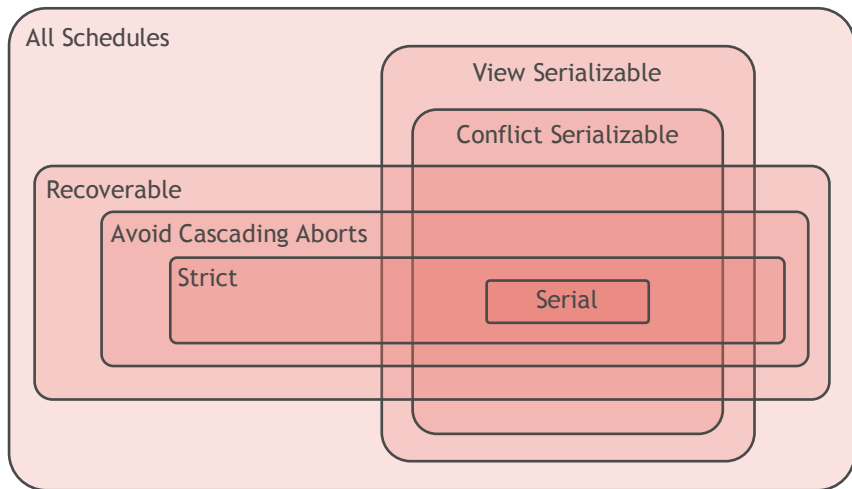
<sup>3</sup>来源: Andy Pavlo, CMU 15-445



## Example (SS2PL)

$T_1$	$T_2$
X-LOCK(A)	
X-LOCK(B)	
r(A)	X-LOCK(A)
w(A)	
r(B)	
w(B)	
UNLOCK(A)	
UNLOCK(B)	
ABORT	
	r(A)
	w(A)
	r(A)

# 调度的分类



# Lock-based Concurrency Control

## Deadlocks

# 死锁(Deadlock)

如果一组事务中每个事务都在等待其他事务释放锁，则这组事务形成死锁

## Example (死锁)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	S-LOCK(B)	
	r(B)	
	S-LOCK(A)	
w(A)		
X-LOCK(B)		
⋮	⋮	

# 死锁(Deadlock)

如果一组事务中每个事务都在等待其他事务释放锁，则这组事务形成死锁

## Example (死锁)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	S-LOCK(B)	将B上的共享锁授予 $T_2$
	r(B)	
	S-LOCK(A)	
w(A)		
X-LOCK(B)		
⋮	⋮	

# 死锁(Deadlock)

如果一组事务中每个事务都在等待其他事务释放锁，则这组事务形成死锁

## Example (死锁)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	S-LOCK(B)	将B上的共享锁授予 $T_2$
	r(B)	
	S-LOCK(A)	拒绝!
w(A)		
X-LOCK(B)		
⋮	⋮	

## 死锁(Deadlock)

如果一组事务中每个事务都在等待其他事务释放锁，则这组事务形成死锁

### Example (死锁)

$T_1$	$T_2$	调度器的动作
X-LOCK(A)		将A上的互斥锁授予 $T_1$
r(A)		
	S-LOCK(B)	将B上的共享锁授予 $T_2$
	r(B)	
	S-LOCK(A)	拒绝!
w(A)		
X-LOCK(B)		拒绝!
⋮	⋮	
		形成死锁!

# 死锁的处理

## 措施1: 死锁检测(deadlock detection)

- DBMS检测死锁是否发生
- 如果发生了死锁，则采取办法解除死锁

## 措施2: 死锁预防(deadlock prevention)

- 预防死锁的发生



# 死锁检测(Deadlock Detection)

## 方法1: 超时检测(timeout)

- 如果在给定的时间内没有任何事务执行，则认为死锁发生

## 方法2: 等待图(waits-for graph)检测

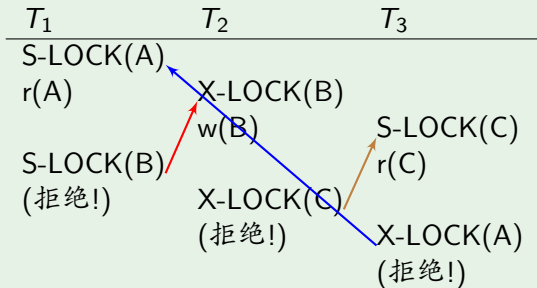
- DBMS用等待图表示事务关于锁的等待关系
- DBMS定期检查等待图中是否存在环(cycle)
- 如果等待图中有环，则死锁发生

# 等待图(Waits-for Graph)

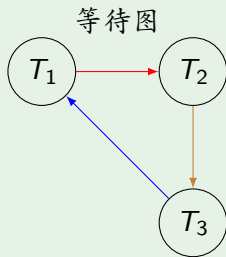
- 顶点代表事务
- 如果事务 $T_i$ 正在等待事务 $T_j$ 释放锁，则存在从 $T_i$ 到 $T_j$ 的有向边(arc)

事务产生死锁当且仅当等待图中有环(cycle)

## Example (等待图)



$T_1, T_2, T_3$  产生死锁



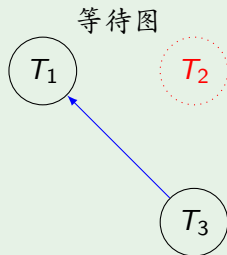
# 死锁解除(Deadlock Resolution)

从等待图的环中选择一个事务作为“牺牲品”，中止该事务

## Example (死锁解除)

$T_1$	$T_2$	$T_3$
S-LOCK(A)		
r(A)	X-LOCK(B)	
	w(B)	S-LOCK(C)
S-LOCK(B)		r(C)
(拒绝!)	X-LOCK(C)	
	(拒绝!)	X-LOCK(A)
		(拒绝!)

ABORT



## 死锁解除(续)

选择“牺牲品”时需要考虑多种因素

- 事务的年龄/启动时间
- 事务的进度(已执行的查询数量)
- 事务获得的锁的数量
- 需要级联中止的事务数量

还要考虑事务“被牺牲”的次数，防止事务“饿死”(starvation)

# 死锁预防(Deadlock Prevention)

当事务启动时，DBMS为事务分配一个唯一且固定的优先级(priority)

- 开始得越早，优先级越高

当事务 $T_i$ 请求事务 $T_j$ 拥有的锁而无法获得时，DBMS根据 $T_i$ 和 $T_j$ 的优先级来裁决如何处理 $T_i$ 和 $T_j$ ，可采用下面两条规则之一来预防

- Wait-Die规则
- Wound-Wait规则

使用上述规则来处理加锁请求，则不会产生死锁

## Wait-Die规则——“尊老爱幼”

事务  $T_i$  请求事务  $T_j$  拥有的锁而无法获得

- 如果  $T_i$  比  $T_j$  的优先级高，则  $T_i$  等待
- 否则， $T_i$  中止 ( $T_i$  重启后，其优先级保持不变(为什么?))

### Example (Wait-Die规则)

设  $T_1$  的优先级高于  $T_2$  的优先级

$T_1$	$T_2$	Wait-Die
X-LOCK(A)	X-LOCK(A)	$T_1$ 等待
$T_1$	$T_2$	Wait-Die
X-LOCK(A)	X-LOCK(A)	$T_2$ 中止

## Wound-Wait规则——“论资排辈”

事务  $T_i$  请求事务  $T_j$  拥有的锁而无法获得

- 如果  $T_i$  比  $T_j$  的优先级高，则  $T_j$  中止( $T_j$  重启后，其优先级保持不变)
- 否则， $T_i$  等待

### Example (Wait-Die规则)

设  $T_1$  的优先级高于  $T_2$  的优先级

$T_1$	$T_2$	Wound-Wait
X-LOCK(A)	X-LOCK(A)	$T_2$ 中止
$T_1$	$T_2$	Wound-Wait
X-LOCK(A)	X-LOCK(A)	$T_2$ 等待

# Lock-based Concurrency Control

## Multi-Granularity Locking



# 锁的效率问题

锁越多，管理锁的开销越大

- 一个事务访问1亿条元组，就需要加1亿把锁

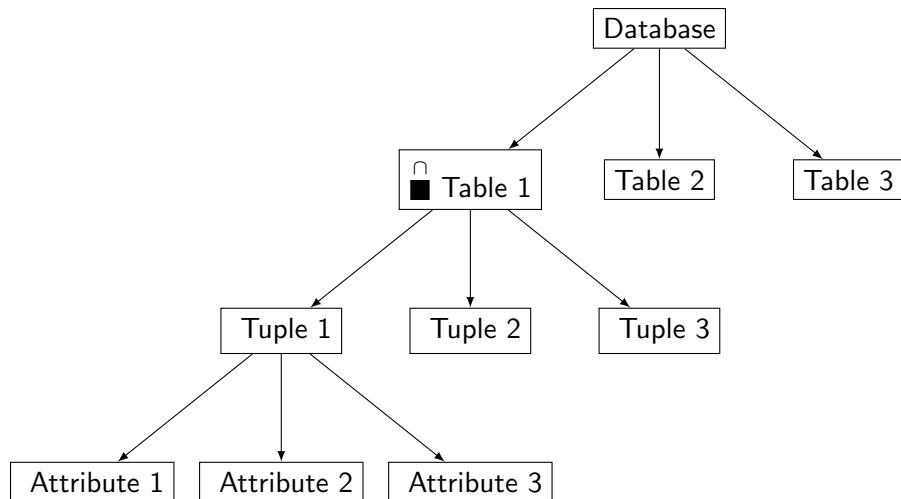
在不合适的粒度上加锁会降低事务并发度

- 一个事务只需访问1条元组，却在整个关系上加锁

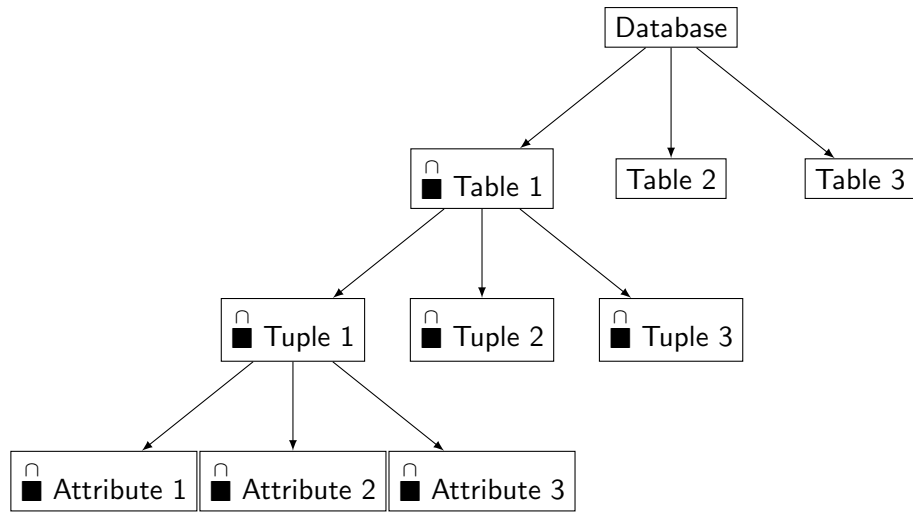
提高锁管理的效率

- 尽可能少加锁
- 在合适的粒度上加锁

# 锁的粒度(Lock Granularity)



# 锁的粒度(Lock Granularity)



# 意向锁(Intension Lock)

## **IS** 意向共享锁(Intension-Shared Locks)/IS锁(IS-Locks)

- 对一个对象加IS锁表示要对该对象的某个(些)后裔加S锁

## **IX** 意向互斥锁(Intension-Exclusive Locks)/IX锁(IX-Locks)

- 对一个对象加IX锁表示要对该对象的某个(些)后裔加X锁或S锁

## **SIX** 共享意向互斥锁(Shared Intension-Exclusive Locks)/SIX锁(SIX-Locks)

- 对一个对象加SIX锁表示要对该对象及其所有后裔加S锁，并且对该对象的某个(些)后裔加X锁

# 相容矩阵(Compatibility Matrix)

		请求锁类型				
		IS	IX	S	SIX	X
拥有锁类型	IS	Yes	Yes	Yes	Yes	No
	IX	Yes	Yes	No	No	No
	S	Yes	No	Yes	No	No
	SIX	Yes	No	No	No	No
	X	No	No	No	No	No

# 多粒度锁协议(Multi-Granularity Locking)

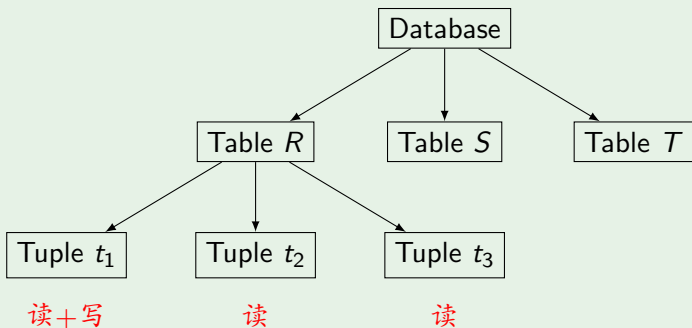
任何事务都要服从下列规则

- ① 从最高级别对象开始加锁，加锁过程自顶向下
- ② 对一个对象加IS或S锁之前，必须先获得其父亲对象的IS锁
- ③ 事务对一个对象加IX、SIX或X锁之前，必须先获得其父亲对象的IX锁
- ④ 解锁过程自底向上

# 多粒度锁协议的示例

## Example (多粒度锁协议)

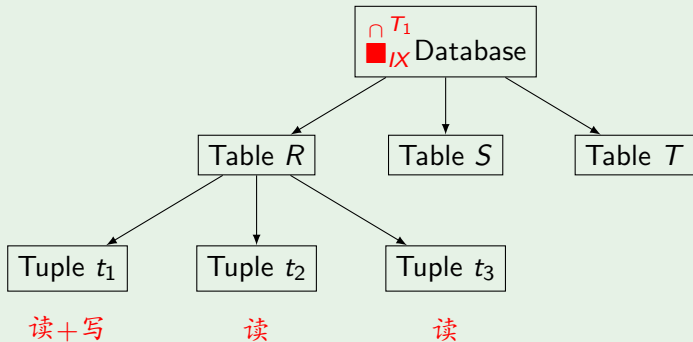
- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$



# 多粒度锁协议的示例

## Example (多粒度锁协议)

- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$

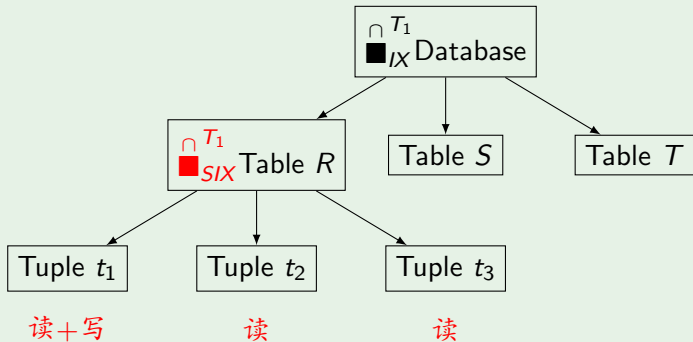




# 多粒度锁协议的示例

## Example (多粒度锁协议)

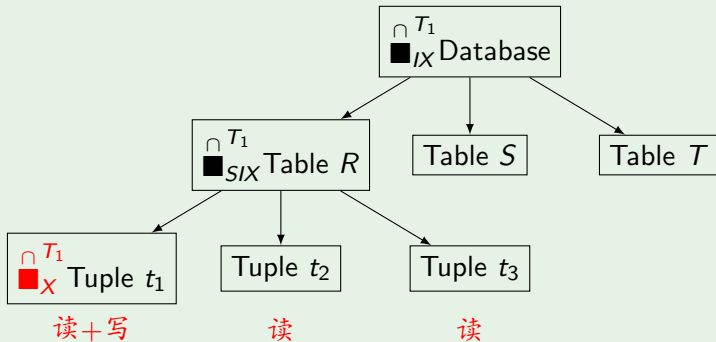
- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$



# 多粒度锁协议的示例

## Example (多粒度锁协议)

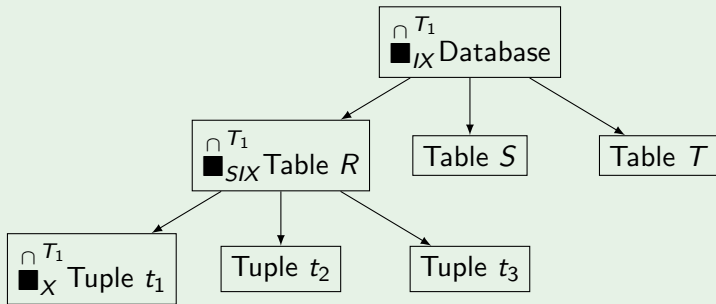
- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$



# 多粒度锁协议的示例

## Example (多粒度锁协议)

- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$

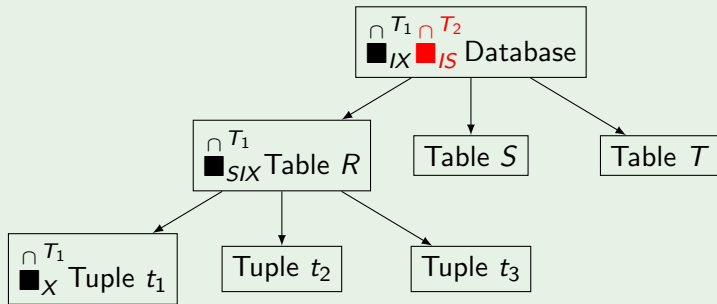


读

# 多粒度锁协议的示例

## Example (多粒度锁协议)

- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$

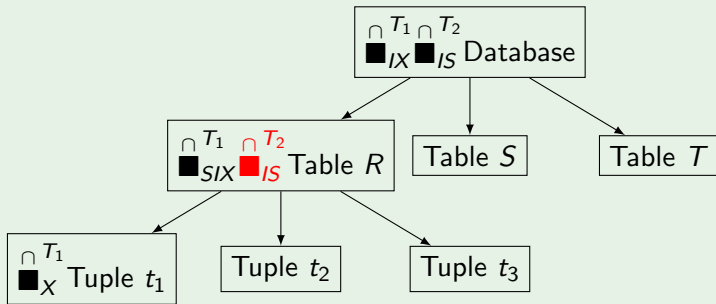


读

# 多粒度锁协议的示例

## Example (多粒度锁协议)

- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$

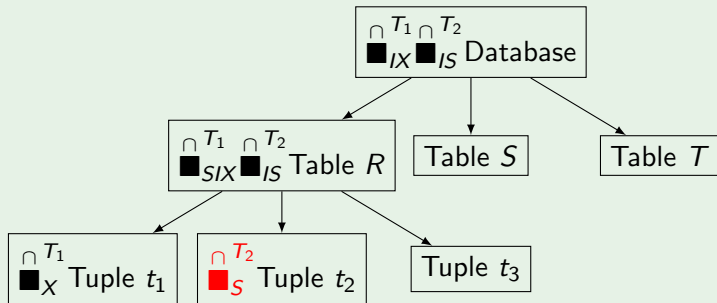


读

# 多粒度锁协议的示例

## Example (多粒度锁协议)

- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$

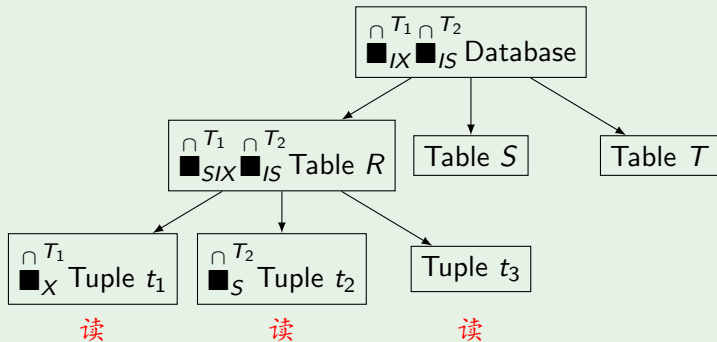


读

# 多粒度锁协议的示例

## Example (多粒度锁协议)

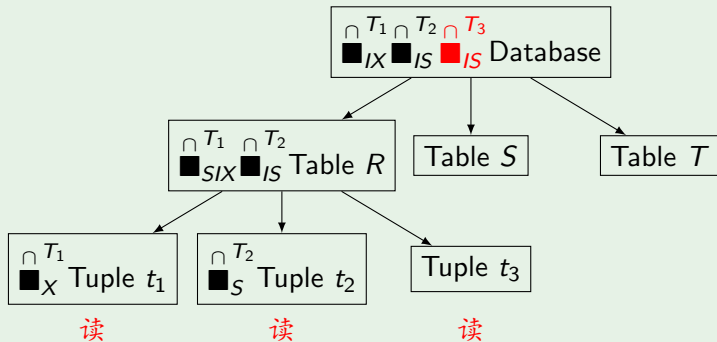
- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$



# 多粒度锁协议的示例

## Example (多粒度锁协议)

- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$

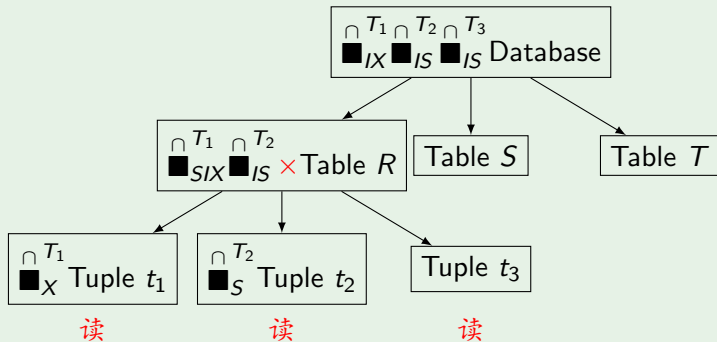




# 多粒度锁协议的示例

## Example (多粒度锁协议)

- 事务  $T_1$  扫描关系  $R$  并修改元组  $t_1$
- 事务  $T_2$  读关系  $R$  的元组  $t_2$
- 事务  $T_3$  扫描关系  $R$  : 对  $R$  加  $S$  锁被拒,  $T_3$  等待



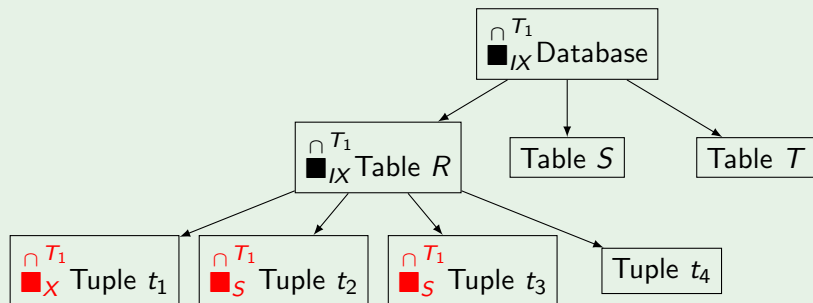
# 锁升级(Lock Escalation)

如果一个事务已经请求了大量低级别对象上的锁，则DBMS会动态地将这些锁升级为上一级别对象上的锁

- 减少锁的数量
- 选择合适的锁粒度

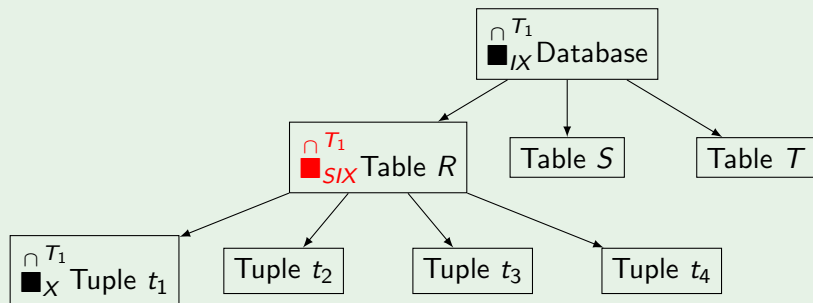
# 锁升级的示例

## Example (锁升级)



# 锁升级的示例

## Example (锁升级)



# Lock-based Concurrency Control

## Phantoms

# 动态数据库(Dynamic Databases)

前面假设事务只执行读操作和更新操作

冲突可串行化调度只在固定的数据库上保证

如果事务还执行插入或删除操作，则可能出现幻读(phantom)

# 幻读(Phantom)

## Example (幻读)

$T_1$	$T_2$	$T_1$ 的结果
BEGIN	BEGIN	
SELECT MAX(val) FROM t		200
WHERE id > 1;		
	INSERT INTO t	
	VALUES (3, 300);	
	COMMIT;	
SELECT MAX(val) FROM t		300
WHERE id > 1;		
COMMIT;		

t	
id	val
1	100
2	200

原因:  $T_1$ 只能锁定t中id > 1的元组, 无法给不存在的元组加锁

# 谓词锁(Predicate Lock)

可以用谓词锁来解决幻读问题

## Example (谓词锁)

$T_1$	$T_2$	调度器的动作
<code>LOCK('id &gt; 1')</code> <code>SELECT MAX(val) FROM t</code> <code>WHERE id &gt; 1;</code>		将锁授予 $T_1$
<code>SELECT MAX(val) FROM t</code> <code>WHERE id &gt; 1;</code> <code>UNLOCK('id &gt; 1')</code> <code>COMMIT</code>	<code>LOCK('id &gt; 1')</code>	拒绝
	<code>INSERT INTO t</code> <code>VALUES (3, 300);</code> <code>UNLOCK('id &gt; 1')</code> <code>COMMIT</code>	释放锁  将锁授予 $T_2$



# Next-Key锁(Next-Key Lock)

谓词锁的实现代价很高

- 谓词的数量远远比对象的数量大
- 谓词之间的关系极其复杂

MySQL使用next-key锁(next-key lock)

# Timestamp Ordering (T/O) Concurrency Control

# 时间戳(Timestamp)

DBMS为每个事务  $T_i$  分配一个唯一且固定的数，作为  $T_i$  的时间戳，记作  $TS(T_i)$

- 时间戳单调递增：分配得越晚，时间戳越大

不同方法分配时间戳的时机不同

- 事务启动时分配，如基本T/O (Basic Timestamp Ordering)
- 事务的验证阶段开始时分配，如乐观并发控制(Optimistic Concurrency Control, OCC)

# 时间戳定序的并发控制(Timestamp Ordering Concurrency Control, T/O)

事务读写对象前无需加锁

根据时间戳来确定事务的可串行化调度

- 如果  $TS(T_i) < TS(T_j)$ , 则DBMS必须确保在事务的调度所等价的串行调度中,  $T_i$  出现在  $T_j$  之前

# Timestamp Ordering (T/O) Concurrency Control

## Basic T/O

# Basic T/O

每个对象 $X$ 关联着2个时间戳

- $RTS(X)$ : 成功读 $X$ 的最晚的事务的时间戳
- $WTS(X)$ : 成功写 $X$ 的最晚的事务的时间戳

对事务的每个读写操作进行时间戳检查

- 如果一个事务准备访问一个“来自未来”的对象，则该事务中止，并重启为一个新事务

# Basic T/O的读操作

事务  $T_i$  准备读对象  $X$

如果  $TS(T_i) < WTS(X)$ , 则

- 中止  $T_i$  并重启  $T_i$  (分配新的时间戳)

否则

- 允许  $T_i$  读  $X$
- 将  $RTS(X)$  更新为  $\max(RTS(X), TS(T_i))$
- 在事务的工作区(workspace)中创建  $X$  的局部副本, 以保证可重复读

# Basic T/O的写操作

事务  $T_i$  准备写对象  $X$

如果  $TS(T_i) < RTS(X)$  或  $TS(T_i) < WTS(X)$ , 则

- 中止并重启  $T_i$

否则

- 允许  $T_i$  写  $X$
- 将  $WTS(X)$  更新为  $\max(WTS(X), TS(T_i))$
- 创建  $X$  的局部副本



# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
$r(B)$	
	BEGIN
	$r(B)$
	$w(B)$
$r(A)$	
	$r(A)$
	$w(A)$
COMMIT	COMMIT

Object	RTS	WTS
A	0	0
B	0	0

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
$r(B)$	
	BEGIN
	$r(B)$
	$w(B)$
$r(A)$	
	$r(A)$
	$w(A)$
COMMIT	COMMIT

Object	RTS	WTS
A	0	0
B	1	0

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
$r(B)$	
	BEGIN
	$r(B)$
	$w(B)$
$r(A)$	
	$r(A)$
	$w(A)$
COMMIT	COMMIT

Object	RTS	WTS
A	0	0
B	1	0

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
r(B)	
	BEGIN
	r(B)
	w(B)
r(A)	
	r(A)
	w(A)
COMMIT	COMMIT

Object	RTS	WTS
A	0	0
B	2	0

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
$r(B)$	
	BEGIN
	$r(B)$
	$w(B)$
$r(A)$	
	$r(A)$
	$w(A)$
COMMIT	COMMIT

Object	RTS	WTS
A	0	0
B	2	0

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
r(B)	
	BEGIN
	r(B)
	w(B)
r(A)	
	r(A)
	w(A)
COMMIT	COMMIT

Object	RTS	WTS
A	0	0
B	2	2

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
r(B)	
	BEGIN
	r(B)
	w(B)
r(A)	
	r(A)
	w(A)
COMMIT	COMMIT

Object	RTS	WTS
A	0	0
B	2	2

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
$r(B)$	
	BEGIN
	$r(B)$
	$w(B)$
$r(A)$	
	$r(A)$
	$w(A)$
COMMIT	COMMIT

Object	RTS	WTS
A	1	0
B	2	2



# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
r(B)	
	BEGIN
	r(B)
	w(B)
r(A)	
	r(A)
	w(A)
COMMIT	COMMIT

Object	RTS	WTS
A	1	0
B	2	2

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
r(B)	
	BEGIN
	r(B)
	w(B)
r(A)	
	r(A)
	w(A)
COMMIT	COMMIT

Object	RTS	WTS
A	2	0
B	2	2

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
r(B)	
	BEGIN
	r(B)
	w(B)
r(A)	
	r(A)
	w(A)
COMMIT	COMMIT

Object	RTS	WTS
A	2	0
B	2	2

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
r(B)	
	BEGIN
	r(B)
	w(B)
r(A)	
	r(A)
	w(A)
COMMIT	COMMIT

Object	RTS	WTS
A	2	2
B	2	2

# Basic T/O 示例1

## Example (Basic T/O 例1)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
$r(B)$	BEGIN
	$r(B)$
	$w(B)$
$r(A)$	$r(A)$
	$w(A)$
COMMIT	COMMIT

Object	RTS	WTS
A	2	2
B	2	2

没有违反时间戳顺序， $T_1$ 和 $T_2$ 都可以提交

# Basic T/O 示例2

## Example (Basic T/O 例2)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN $r(A)$	BEGIN $w(A)$ COMMIT
$w(A)$	
$r(A)$	
COMMIT	

Object	RTS	WTS
A	0	0
B	0	0

# Basic T/O 示例2

## Example (Basic T/O 例2)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN $r(A)$	BEGIN $w(A)$ COMMIT
$w(A)$ $r(A)$ COMMIT	

Object	RTS	WTS
A	1	0
B	0	0

# Basic T/O 示例2

## Example (Basic T/O 例2)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN $r(A)$	BEGIN $w(A)$ COMMIT
$w(A)$ $r(A)$ COMMIT	

Object	RTS	WTS
A	1	0
B	0	0



# Basic T/O 示例2

## Example (Basic T/O 例2)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN $r(A)$	BEGIN $w(A)$ COMMIT
$w(A)$ $r(A)$ COMMIT	

Object	RTS	WTS
A	1	2
B	0	0

# Basic T/O 示例2

## Example (Basic T/O 例2)

调度

$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN $r(A)$	BEGIN $w(A)$ COMMIT
$w(A)$	
$r(A)$	
COMMIT	

Object	RTS	WTS
A	1	2
B	0	0

# Basic T/O 示例2

## Example (Basic T/O 例2)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN	
$r(A)$	
	BEGIN
	$w(A)$
	COMMIT
$w(A)$	
$r(A)$	
COMMIT	

Object	RTS	WTS
A	1	2
B	0	0

$T_1$  不能覆盖  $T_2$  修改过的 A, 故  $T_1$  中止并重启

# Thomas写规则(Thomas Write Rule)

事务  $T_i$  准备写对象  $X$

如果  $TS(T_i) < RTS(X)$ , 则

- 中止  $T_i$  并重启  $T_i$  (分配新的时间戳)

如果  $TS(T_i) < WTS(X)$ , 则

- 忽略这个写操作并允许  $T_i$  继续执行(Thomas写规则)
- 这违反了  $T_i$  的时间戳顺序

否则

- 允许  $T_i$  写  $X$
- 将  $WTS(X)$  更新为  $\max(WTS(X), TS(T_i))$

# Thomas写规则示例

## Example (Thomas写规则)

调度	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN $r(A)$	BEGIN $w(A)$ COMMIT
$w(A)$	
$r(A)$	
COMMIT	

Database		
Object	RTS	WTS
A	1	2
B	0	0

应用Thomas写规则：忽略 $T_1$ 对A的写操作，继续执行 $T_1$

# Basic T/O的性质

如果不使用Thomas写规则，则Basic T/O生成冲突可串行化调度

- 因为任何事务都无需等待，所以无死锁
- 如果某个长事务(long txn)总是和其他短事务(short txn)发生冲突，则这个长事务可能会“饿死(starve)”(不断中止并重启)

Basic T/O可能生成不可恢复的调度(unrecoverable schedule)

## 不可恢复调度(Unrecoverable Schedule)

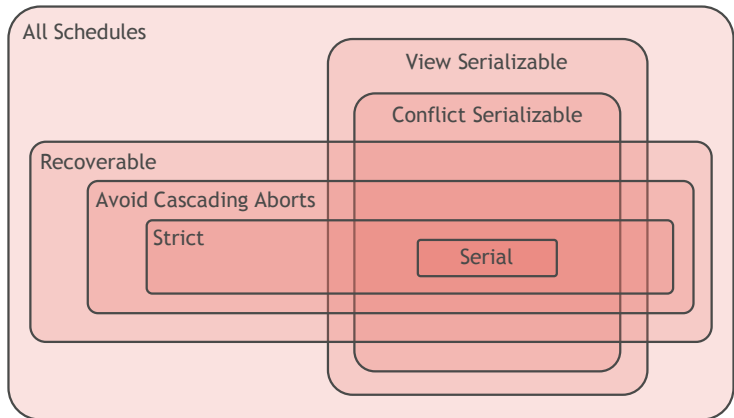
Schedule	
$T_1$	$T_2$
$TS(T_1) = 1$	$TS(T_2) = 2$
BEGIN $w(A)$	
	BEGIN $r(A)$ $w(B)$ COMMIT
Crash! COMMIT	

该调度不可恢复

- 在故障恢复时, DBMS undo  $T_1$
- 因为  $T_2$  读了  $T_1$  修改过的  $A$ , 所以必须 undo  $T_2$
- 但  $T_2$  已提交, 故无法 undo  $T_2$

# 可恢复调度(Recoverable Schedule)

A schedule is **recoverable** if every transaction in that schedule commits only after all the transactions whose changes it reads have committed.





# Basic T/O的不足

开销大

- 必须将数据复制到事务的工作区(workspace)
- 频繁更新时间戳

长事务可能会饿死

# Timestamp Ordering (T/O) Concurrency Control

## Optimistic Concurrency Control (OCC)

# 乐观并发控制(Optimistic Concurrency Control, OCC)

## 基本假设

- 并发事务的冲突很少
- 大多数事务都很短

## OCC中事务的三个执行阶段

- ① 读阶段(read phase)
- ② 验证阶段(validation phase)
- ③ 写阶段(write phase)

# 读阶段(Read Phase)

- DBMS为事务创建私有工作区(private workspace)
- 事务读的每个对象都要复制到工作区，以保证可重复读
- 事务对对象的修改写到工作区，而不是数据库
- 将读和写分别记录到读集合(read set)和写集合(write set)中

# 验证阶段(Validation Phase)

- 当验证阶段开始时，DBMS为事务分配时间戳
- DBMS对事务的写集合(write set)进行检查，判断该事务是否与其他活跃事务(active txn)冲突

## 写阶段(Write Phase)

- 如果没有冲突，则将写集合(write set)写回数据库
- 否则，中止该事务，并重启为新事务

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read	
r(A)	//Read
	r(A)
	//Validate
	//Write
	COMMIT
w(A)	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read $r(A)$	//Read $r(A)$
	//Validate
	//Write
	COMMIT
$w(A)$	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS



# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read $r(A)$	//Read $r(A)$
	//Validate
	//Write
	COMMIT
$w(A)$	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	123	0

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read	
r(A)	<b>//Read</b>
	r(A)
	//Validate
	//Write
	COMMIT
w(A)	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	123	0

$T_2$ Workspace		
Object	Value	WTS

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read $r(A)$	//Read $r(A)$
	//Validate
	//Write
	COMMIT
$w(A)$	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	123	0

$T_2$ Workspace		
Object	Value	WTS
A	123	0

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read $r(A)$	//Read $r(A)$
	//Validate $TS(T_2) = 1$
	//Write COMMIT
$w(A)$ //Validate //Write COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	123	0

$T_2$ Workspace		
Object	Value	WTS
A	123	0

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read	
r(A)	//Read
	r(A)
	//Validate
	$TS(T_2) = 1$
	//Write
	COMMIT
w(A)	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	123	0

$T_2$ Workspace		
Object	Value	WTS
A	123	0

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read	
r(A)	//Read
	r(A)
	//Validate
	$TS(T_2) = 1$
	//Write
	COMMIT
w(A)	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	123	0

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read	
r(A)	//Read
	r(A)
	//Validate
	$TS(T_2) = 1$
	//Write
	COMMIT
w(A)	
//Validate	
//Write	
COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	456	$\infty$

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read $r(A)$	//Read $r(A)$
	//Validate $TS(T_2) = 1$
	//Write COMMIT
$w(A)$ //Validate $TS(T_1) = 2$ //Write COMMIT	

Database		
Object	Value	WTS
A	123	0

$T_1$ Workspace		
Object	Value	WTS
A	456	$\infty$



# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read $r(A)$	//Read $r(A)$
	//Validate $TS(T_2) = 1$
	//Write COMMIT
$w(A)$ //Validate $TS(T_1) = 2$ //Write COMMIT	

Database		
Object	Value	WTS
A	456	2

$T_1$ Workspace		
Object	Value	WTS
A	456	$\infty$

# OCC 示例

## Example (OCC 示例)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
//Read $r(A)$	//Read $r(A)$
	//Validate $TS(T_2) = 1$
	//Write COMMIT
$w(A)$ //Validate $TS(T_1) = 2$ //Write COMMIT	

Database		
Object	Value	WTS
A	456	2

## 验证阶段(Validation Phase)

事务  $T_j$  进入验证阶段后

DBMS 为  $T_j$  分配时间戳  $TS(T_j)$

对任意  $TS(T_i) < TS(T_j)$  的事务  $T_i$ ，如果下面3个条件之一满足，则  $T_j$  与  $T_i$  不冲突

- 条件1: 在  $T_j$  开始前， $T_i$  已完成3个阶段
- 条件2: 在  $T_j$  的写阶段开始前， $T_i$  已完成3个阶段，且  $T_i$  写的对象均未被  $T_j$  读过，即  $WS(T_i) \cap RS(T_j) = \emptyset$
- 条件3: 在  $T_j$  的读阶段完成前， $T_i$  的读阶段已完成，且  $T_i$  写的对象均未被  $T_j$  读写过，即  $WS(T_i) \cap RS(T_j) = \emptyset$  且  $WS(T_i) \cap WS(T_j) = \emptyset$
- 每个条件都保证了  $T_j$  的写对  $T_i$  不可见

如果所有满足  $TS(T_i) < TS(T_j)$  的事务  $T_i$  与  $T_j$  都不冲突，则  $T_j$  通过验证，可以进入写阶段

## 验证条件1

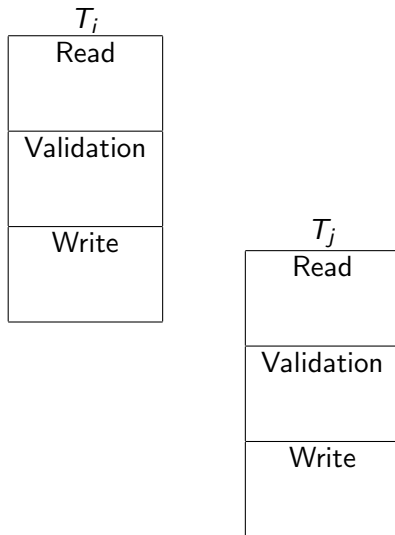
在  $T_j$  开始前,  $T_i$  已完成3个阶段

Schedule	
$T_i$	$T_j$
BEGIN	
//Read	
//Validate	
//Write	
COMMIT	
	BEGIN
	//Read
	//Validate
	//Write
	COMMIT

该调度就是由时间戳确定的串行调度

## 验证条件2

在  $T_j$  的写阶段开始前,  $T_i$  已完成3个阶段, 且  $WS(T_i) \cap RS(T_j) = \emptyset$



## Example (满足验证条件2的调度)

Schedule	
$T_i$	$T_j$
BEGIN	BEGIN
	//Read
	r(A)
//Read	w(A)
r(A)	
//Validate	
//Write	
COMMIT	
	//Validate
	//Write
	COMMIT

$WS(T_i) \cap RS(T_j) = \emptyset$ , 故该调度等价于由时间戳确定的串行调度

## Example (违反验证条件2的调度)

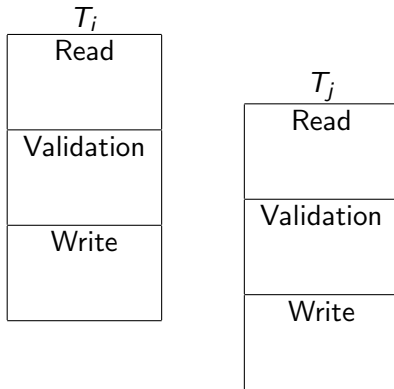
Schedule	
$T_i$	$T_j$
BEGIN	BEGIN
//Read $r(A)$ $w(A)$	
	//Read $r(A)$
//Validate //Write COMMIT	
	//Validate //Write COMMIT

$WS(T_i) \cap RS(T_j) = \{A\}$ , 故该调度不等价于由时间戳确定的串行调度

- 在由时间戳确定的串行调度中,  $T_j$ 读到的 $A$ 应当是由 $T_i$ 写的
- 在该调度中,  $T_j$ 读到的 $A$ 并不是 $T_i$ 写的, 因为当 $T_j$ 读 $A$ 时,  $T_i$ 尚未将修改过的 $A$ 写回数据库

## 验证条件3

在  $T_j$  的读阶段完成前,  $T_i$  的读阶段已完成,  
且  $WS(T_i) \cap RS(T_j) = \emptyset$ ,  $WS(T_i) \cap WS(T_j) = \emptyset$





## Example (满足验证条件3的调度)

Schedule	
$T_i$	$T_j$
BEGIN	BEGIN
//Read $r(A)$ $w(A)$	
	//Read $r(B)$
//Validate	
	$w(B)$ //Validate //Write COMMIT
//Write COMMIT	

$WS(T_i) \cap RS(T_j) = \emptyset$  且  $WS(T_i) \cap WS(T_j) = \emptyset$ , 故该调度等价于由时间戳确定的串行调度

## Example (违反验证条件3的调度)

Schedule	
$T_i$	$T_j$
BEGIN	BEGIN
//Read $r(A)$ $w(A)$	
	//Read $r(A)$
//Validate	
	$r(B)$ //Validate //Write COMMIT
//Write COMMIT	

$WS(T_i) \cap RS(T_j) = \{A\}$ , 故该调度不等价于由时间戳确定的串行调度

- 在由时间戳确定的串行调度中,  $T_j$ 读到的 $A$ 应当是由 $T_i$ 写的
- 在该调度中,  $T_j$ 读到的 $A$ 并不是 $T_i$ 写的, 因为当 $T_j$ 读 $A$ 时,  $T_i$ 尚未将修改过的 $A$ 写回数据库

# 串行验证(Serial Validation)

在临界区(critical section)中内执行事务的验证阶段和写阶段

- 任何时刻只有一个事务处于验证/写阶段
- 当一个事务正在验证时，其他事务不得提交
- 如果一个事务已经完成验证阶段，则只有在其写阶段完成后，其他事务才能开始验证

# OCC的特性

在并发事务冲突很少的情况下，OCC很有效

- 例如，事务都是只读的
- 例如，事务访问的数据子集互不相交

# OCC的不足

- 将数据复制到工作区的开销很大
- 验证阶段和写阶段是瓶颈
- OCC比2PL的事务中止代价高，因为中止事务的读阶段已经完成

# Basic T/O和OCC的区别

## Basic T/O

- Basic T/O在执行事务的每个读写操作前都要进行时间戳检查
- 事务的运行时代价比较高

## OCC

- OCC在执行事务的读写操作前不需要进行时间戳检查，只需在验证阶段进行检查
- 事务中止的代价比较高

# Multi-Version Concurrency Control (MVCC)

# 多版本并发控制(Multi-Version Concurrency Control, MVCC)

DBMS为数据库中每个(逻辑)对象维护多个(物理)版本(version)

- 当事务写一个对象时，DBMS为该对象创建一个新的版本
- 当事务读一个对象时，它读的是该事务启动时已存在的由已提交事务写入的最新版本



# MVCC的特点

**写不阻塞读(Writers don't block readers)**

**读不阻塞写(Readers don't block writers)**

只读(read-only)的事务直接在一致性快照(consistent snapshot)上读，无需加锁

# MVCC 示例

## Example (MVCC 示例1)

Schedule	
$T_1$	$T_2$
BEGIN	
r(A)	
	BEGIN
	w(A)
r(A)	
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active

# MVCC 示例

## Example (MVCC 示例1)

Schedule	
$T_1$	$T_2$
BEGIN	
$r(A) \leftarrow A_0$	
	BEGIN
$r(A)$	$w(A)$
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active

# MVCC 示例

## Example (MVCC 示例1)

Schedule	
$T_1$	$T_2$
BEGIN	
$r(A)$	
	BEGIN
$r(A)$	$w(A)$
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active
$T_2$	2	Active

# MVCC 示例

## Example (MVCC 示例1)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN $w(A)$  COMMIT
$r(A)$	
$r(A)$	
COMMIT	

Database			
Version	Value	Begin	End
$A_0$	123	0	2
$A_1$	456	2	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active
$T_2$	2	Active

# MVCC 示例

## Example (MVCC 示例1)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN $w(A)$
$r(A)$	
$r(A) \leftarrow A_0$	
COMMIT	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	2
$A_1$	456	2	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active
$T_2$	2	Active

# MVCC 示例

## Example (MVCC 示例1)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN $w(A)$
$r(A)$	
$r(A)$	
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	2
$A_1$	456	2	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Committed
$T_2$	2	Committed

# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	
r(A)	
w(A)	BEGIN
	r(A)
	w(A)
r(A)	
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active



# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
$r(A) \leftarrow A_0$	
$w(A)$	
$r(A)$	$r(A)$
COMMIT	$w(A)$
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active

# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
r(A)	
w(A)	
r(A)	
COMMIT	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active

# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	
r(A)	
w(A)	BEGIN
	r(A)
r(A)	w(A)
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active
$T_2$	2	Active

# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
$r(A)$	
$w(A)$	
$r(A)$	$r(A) \leftarrow A_0$
COMMIT	$w(A)$
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active
$T_2$	2	Active

# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
r(A)	
w(A)	
r(A)	w(A) Stall...
COMMIT	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active
$T_2$	2	Active

# MVCC 示例

## Example (MVCC 示例2)

Schedule	
$T_1$	$T_2$
BEGIN	
r(A)	
w(A)	
	BEGIN
	r(A)
	w(A) Stall...
$r(A) \leftarrow A_1$	
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Active
$T_2$	2	Active

# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
r(A)	
w(A)	
r(A)	w(A) Stall...
COMMIT	
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Committed
$T_2$	2	Active

# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN
r(A)	
w(A)	
r(A)	r(A)
COMMIT	w(A)
	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	2
$A_2$	789	2	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Committed
$T_2$	2	Active



# MVCC示例

## Example (MVCC示例2)

Schedule	
$T_1$	$T_2$
BEGIN	BEGIN r(A) w(A)
r(A)	
w(A)	
r(A)	
COMMIT	COMMIT

Database			
Version	Value	Begin	End
$A_0$	123	0	1
$A_1$	456	1	2
$A_2$	789	2	-

Txn Status Table		
TxnId	TS	Status
$T_1$	1	Committed
$T_2$	2	Committed

# 并发控制协议(Concurrency Control Protocol)

写不阻塞读  
读不阻塞写  
写阻塞写

采用并发控制协议解决写-写冲突问题

- 时间戳定序并发控制(Timestamp Ordering, T/O)
- 乐观并发控制(Optimistic Concurrency Control, OCC)
- 两阶段锁协议(Two-Phase Locking, 2PL)

# 总结

- 1 Transactions
- 2 Concurrency Control
  - Schedules
  - Isolation Levels
  - Serializability
- 3 Lock-based Concurrency Control
  - Locks
  - Two-Phase Locking (2PL)
  - Deadlocks
  - Multi-Granularity Locking
  - Phantoms
- 4 Timestamp Ordering (T/O) Concurrency Control
  - Basic T/O
  - Optimistic Concurrency Control (OCC)
- 5 Multi-Version Concurrency Control (MVCC)