



## 事务管理(并发控制技术3)

# 主要学习目标

- 时间戳排序协议
- 快照隔离



# 思考问题

- 不用锁如何才能实现多事务正确并发执行呢？

- 目前所学的封锁协议，每一对冲突事务的次序是在执行时由二者都申请的，由类型不相容的第一个锁决定可串行化次序。
- 另一种决定可串行化次序的方法是事先选定事务的次序。其中最常用的方法是时间戳排序机制。

- 对于系统中每一个事务 $T_i$ ，我们把一个唯一的固定时间戳和它联系起来，此时间戳记为 $TS(T_i)$ 。
- 该时间戳是在事务 $T_i$ 开始执行前由数据库系统赋予的。
- 若事务 $T_i$ 已赋予时间戳 $TS(T_i)$ ，此时有一个新事务 $T_j$ 进入系统，则 $TS(T_i) < TS(T_j)$ 。






➤ 实现这种机制可以采用下面两个简单的方法：

➤ 使用系统时钟的值作为时间戳；即，事务的时间戳等于该事务进入系统时的时钟值。

➤ 使用逻辑计数器，每赋予一个时间戳，计数器增加计数；即，事务的时间戳等于该事务进入系统时的计数器值。



- 在时间戳机制中，每个数据项Q需要和以下两个重要的时间戳相关联：
  - W-TS(Q)：表示当前已成功执行write(Q)的所有事务的最大时间戳；
  - R-TS(Q)：表示当前已成功执行read(Q)的所有事务的最大时间戳。
- 每当有新的read(Q)或write(Q)指令成功执行，这两个时间戳就被更新。

- 事务的时间戳决定了调度中事务串行化的顺序。
- 即，若  $TS(T_i) < TS(T_j)$ ，则DBMS必须保证所产生的调度等价于  $T_i$  出现在  $T_j$  之前的某个串行调度。



## 时间戳排序协议

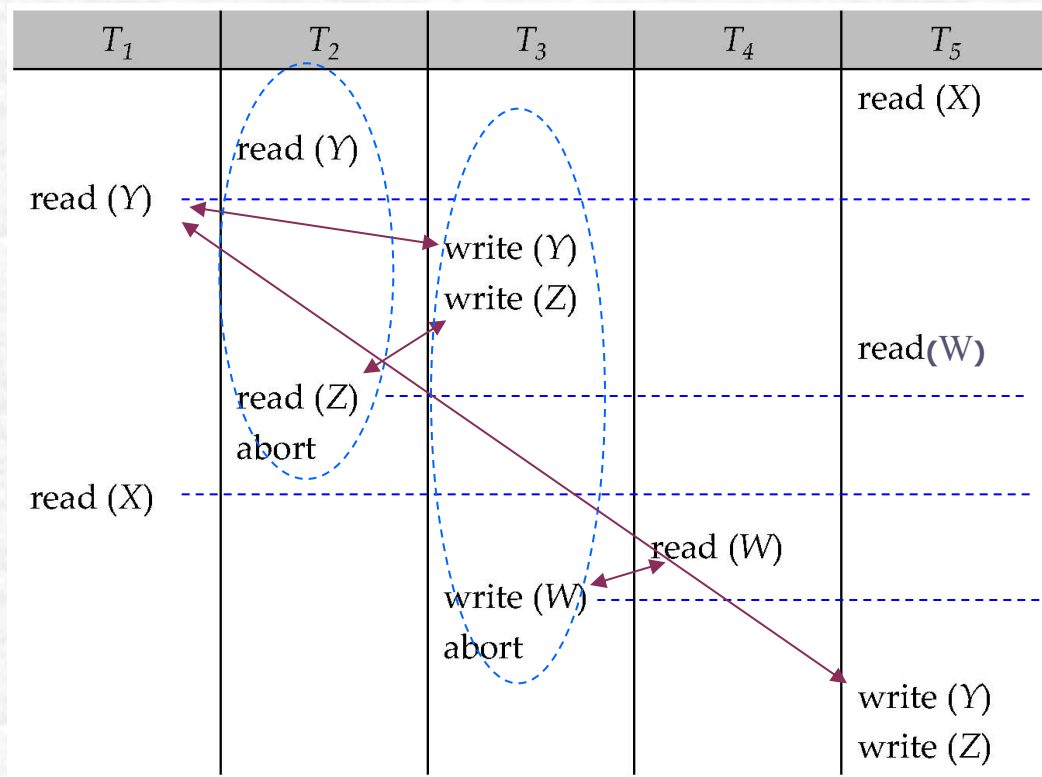
- 时间戳排序协议是按照事务的时间戳顺序来处理事务之间的冲突操作；
- 满足该协议的任何调度都能保证
  - 冲突可串行化的；
  - 无死锁，因为冲突的事务被回滚重启并赋予新的时间戳，而不是等待执行。

## 运作方式

- 假设事务 $T_i$ 发出 $\text{read}(Q)$ 操作（事务时间戳小于冲突操作时间戳时回滚）：
  - 若 $\text{TS}(T_i) < W\text{-TS}(Q)$ ，则 $T_i$ 需要读入的 $Q$ 值已被覆盖。因此， $\text{read}$ 操作被拒绝， $T_i$ 回滚；
  - 若 $\text{TS}(T_i) \geq W\text{-TS}(Q)$ ，则执行 $\text{read}$ 操作，而 $R\text{-TS}(Q)$ 的值被设为 $R\text{-TS}(Q)$ 与 $\text{TS}(T_i)$ 中的较大者。
- 假设事务 $T_i$ 发出 $\text{write}(Q)$ 操作：
  - 若 $\text{TS}(T_i) < R\text{-TS}(Q)$ ，则 $T_i$ 产生的 $Q$ 值是先前所需要的值，但系统已假定该值不会被产生。因此， $\text{write}$ 操作被拒绝， $T_i$ 回滚；
  - 若 $\text{TS}(T_i) < W\text{-TS}(Q)$ ，则 $T_i$ 产生的 $Q$ 值已过时。因此， $\text{write}$ 操作被拒绝， $T_i$ 回滚；
  - 其他情况，系统执行 $\text{write}$ 操作，并将 $W\text{-TS}(Q)$ 的值设为 $\text{TS}(T_i)$ 。

# 时间戳排序协议

一调度如下图, 时间戳分别为 1, 2, 3, 4, 5



$TS(T_i)=i, i=1, \dots, 5$

执行过程分析?

R-timestamp(X)=5 正常读  
R-timestamp(Y)=2 正常读  
R-timestamp(Y)=2 正常读(不更新)  
W-timestamp(Y)=3 正常写(需检查)  
W-timestamp(Z)=3 正常写  
R-timestamp(W)=5 正常读  
异常读, 撤消事务T2(需检查)  
R-timestamp(X)=5 正常读(不更新)  
R-timestamp(W)=5 正常读(不更新)  
异常读, 撤消事务T3(需检查)  
W-timestamp(Y)=5 正常写(需检查)  
W-timestamp(Z)=5 正常写

因通过时间戳协议控制执行, 不再需要加锁!

案例1

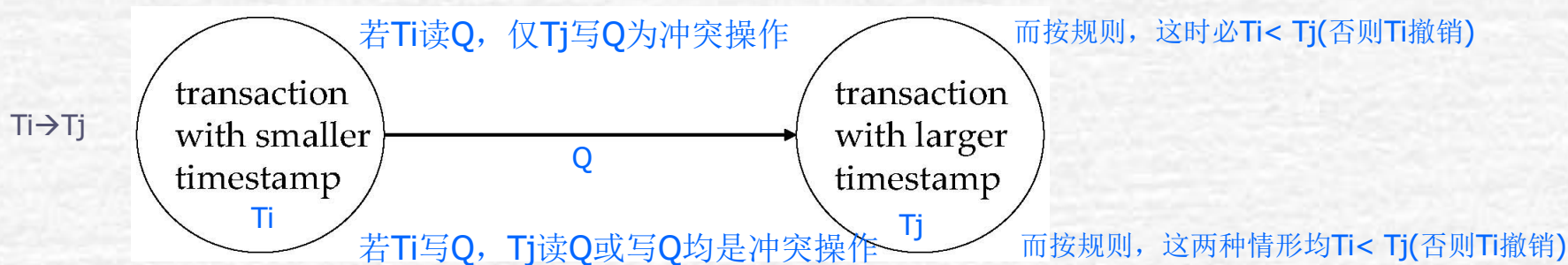
**T1, T4, T5**三个事务正常执行,  
而**T2, T3**两个事务回滚重做。

# 基于时间戳排序协议

2) 时间戳排序协议有什么特点?

为**一有向图**：顶点代表事务，每条弧描述一个冲突操作； $T_i \rightarrow T_j$ 表示 $T_i$ 和 $T_j$ 为冲突操作，且 $T_i$ 先于 $T_j$ 。

- 时间戳排序协议保证了可串行化。
- 优先图中的弧都是从小时间戳到大时间戳：



- 因此，在调度优先图中没有环。

(优先图中无环)

- 1) 时间戳排序协议不仅保证：冲突可串行化！(因为冲突操作按时间戳顺序进行处理)
- 2) 时间戳排序协议还保证：不会出现死锁现象！(因为不要求事务加锁，也不需等待)



时间戳协议的  
调度可恢复吗？

## 基于时间戳排序协议

3) 时间戳排序协议  
有什么缺点？如何  
解决？

一调度如下图，时间戳分别为 1, 2, 3, 4, 5

$$TS(T_i) = i, i = 1 - 5$$

$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read (Y)	read (Y)	write (Y) write (Z)		read (X)
	read (Z) abort			read (Z)
read (X)		write (W) abort	read (W)	write(Z) <b>commit</b>

T5读了T3的数据

$T_j$   $T_i$

可恢复的调度—  
如果事务 $T_j$ 读取了先前由事务 $T_i$   
所写的的数据，  
事务 $T_i$ 需要在 $T_j$ 之前提交

若T3撤消，因T5已提交，导致不可恢复！

案例2



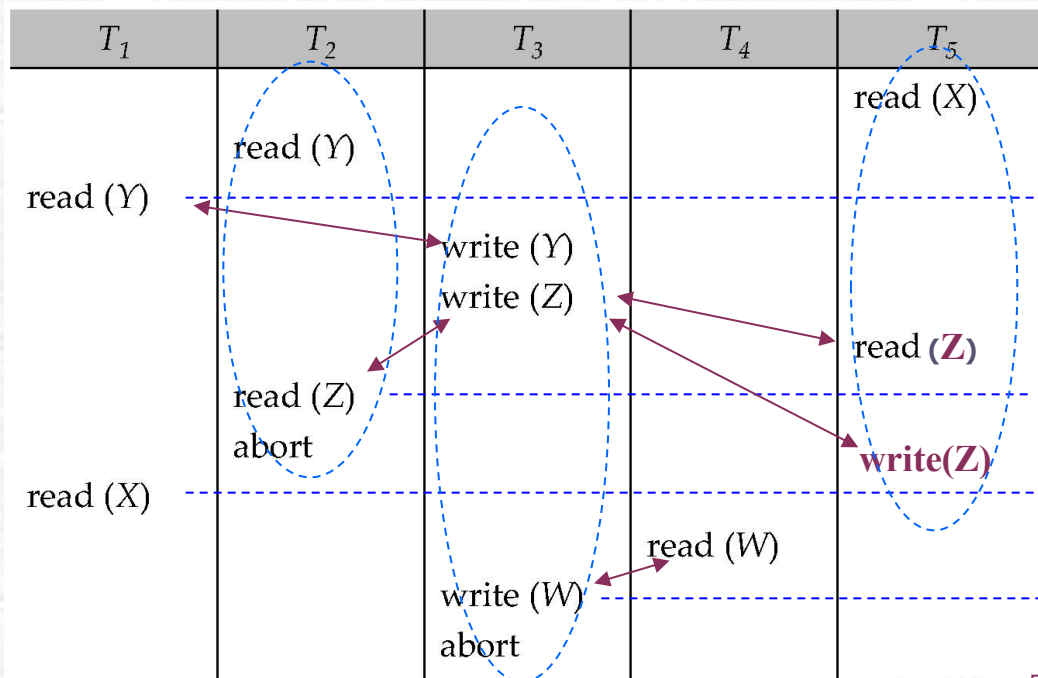
时间戳协议能防止级联回滚吗？

# 基于时间戳排序协议

一调度如下图, 时间戳分别为 1, 2, 3, 4, 5

$$TS(T_i)=i, i=1 - 5$$

### 执行过程分析:



$R\text{-timestamp}(X) = 5$  正常读  
 $R\text{-timestamp}(Y) = 2$  正常读  
 $R\text{-timestamp}(Y) = 2$  正常读 (不更新)  
 $W\text{-timestamp}(Y) = 3$  正常写 (需检查)  
 $W\text{-timestamp}(Z) = 3$  正常写  
 $R\text{-timestamp}(Z) = 5$  正常读 (需检查)  
 异常读, 撤消事务T2 (需检查)  
 $W\text{-timestamp}(Z) = 5$  正常写 (需检查)  
 $R\text{-timestamp}(X) = 5$  正常读 (不更新)  
 $R\text{-timestamp}(W) = 5$  正常读 (不更新)  
 异常读, 撤消事务T3 (需检查)

## 因T3撤消,T5级联回滚!

# 快照隔离

1. 快照隔离的基本思想是什么？

- 快照隔离是在事务开始执行时给它数据库的一份快照。
  - 事务在该快照上操作，和其他并发事务完全隔离。
  - 快照中的数据值仅包括已经提交的事务所写的值。
  - 不能保证可串行化！
- 
- 对只读事务来说是理想的，不需要等待
  - 更新事务，需要在更新写入数据库之前，处理与其他并发更新的事务之间存在的潜在冲突。

# 快照隔离

2. 基于快照隔离，更新事务是如何操作的？

按照先提交者获胜 (first committer wins) 方法，当事务  $T$  进入部分提交状态，以下操作作为一个原子操作执行：

- 检查是否有与  $T$  并发执行的事务，对于  $T$  打算写入的某些数据，该事务已经将更新写入数据库。
- 如果发现这样的事务，则  $T$  中止。
- 如果没有发现这样的事务，则  $T$  提交，并且将更新写入数据库。

按照先更新者获胜 (first updater wins) 方法，系统采用一种仅用于更新操作的锁机制 (读操作不受此影响，因为它们不获得锁)。当事务  $T_i$  试图更新一个数据项时，它请求该数据项的一个写锁。如果没有另一个并发事务持有该锁，则获得锁后执行以下步骤：

- 如果这个数据项已经被任何并发事务更新，则  $T_i$  中止。
- 否则  $T_i$  能够执行其操作，可能包括提交。

然而，如果另一个并发事务  $T_j$  已经持有该数据项的写锁，则  $T_i$  不能执行，并且执行以下规则：

- $T_i$  等待直到  $T_j$  中止或提交。
  - 如果  $T_j$  中止，则锁被释放并且  $T_i$  可以获得锁。当获得锁后，执行前面描述的对于并发事务的更新检查：如果有另一个并发事务更新过该数据项，则  $T_i$  中止；否则，执行其操作。
  - 如果  $T_j$  提交，则  $T_i$  必须中止。

当事务提交或中止时，锁被释放。



# 基于有效性检查的协议

## Validation-Based Protocol

1. 基于有效性检查的协议的阶段?

- In this protocol, execution of each transaction  $T_i$  is done in **three phases**.
  1. **Read and execution phase**: Transaction  $T_i$  writes only to **temporary local variables**
  2. **Validation phase**: Transaction  $T_i$  performs a "validation test" to **determine** if local variables can be written without violating serializability.
  3. **Write phase**: If  $T_i$  is validated, the **updates** are applied to the database; otherwise,  $T_i$  is rolled back.
- The three phases of concurrently executing transactions can be **interleaved** 交替, but each transaction **must go** through the three phases in that order.
  - **Assume** for simplicity that the validation and write phase occur **together**, **atomically and serially**
    - I.e., only one transaction executes validation/write at a time.
- Also called as **optimistic** 乐观 **concurrency control** since transaction executes fully in the **hope** that all will go well during validation  
(而封锁协议和时间戳排序协议等, 称为悲观并发协议, 他们检查到冲突时, 或强迫事务等待或回滚)

# 基于有效性检查的协议

2. 基于有效性检查的协议是如何实现, 适用于什么情况?

## Validation-Based Protocol (Cont.)

- Each transaction  $T_i$  has **three** timestamps
  - **Start( $T_i$ )** : the time when  $T_i$  started its execution
  - **Validation( $T_i$ )** : the time when  $T_i$  entered its validation phase
  - **Finish( $T_i$ )** : the time when  $T_i$  finished its write phase
- Serializability order is **determined** by timestamp given at **validation time**, to increase concurrency.
  - Thus  $TS(T_i)$  is given the value of  $Validation(T_i)$ .
- This protocol is **useful** and gives greater degree of concurrency if probability of conflicts is **low**.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

合法性检查应满足:  
若  $TS(T_i) < TS(T_k)$ , 等价  
的串行调度应满足  
 $T_i$  在  $T_k$  之前完成效果.

为可串行化调度!

即令:  $TS(T_i) = Validation(T_i)$ .

该协议特别适合  
场合?

协议适合情况:  
大部分事务都是  
只读事务, 事务发  
生冲突的频率低!



# 基于有效性检查的协议

☆如何进行事务  
有效性检查？

## Validation Test for Transaction $T_j$

- If for **all**  $T_i$  with  $TS(T_i) < TS(T_j)$  **either one** of the following condition holds:
  - 1)  $finish(T_i) < start(T_j)$ ;
  - 2)  $start(T_j) < finish(T_i) < validation(T_j)$ , **and** the set of data items written by  $T_i$  does **not intersect**(不相交) with the set of data items read by  $T_j$ .**then** validation succeeds and  $T_j$  can be **committed**. Otherwise, validation fails and  $T_j$  is aborted.
- 可串行化说明:
  - 1)  $T_j$ 读数据(和写数据, **start**阶段之后的**finish**阶段)均在 $T_i$ 全部完成写数据之后, 故可按 $T_i$ 、 $T_j$ 次序(冲突)串行化。
  - 2) 首先注意:  $T_j$ 写数据(validation之后的**finish**阶段)晚于 $T_i$ 写数据(**finish**阶段); 然后, 虽然 $T_j$ 读数据(**start**阶段)早于 $T_i$ 写数据(**finish**阶段), 但两数据项集不相交, 因而不会发生冲突操作, 故仍然可按 $T_i$ 、 $T_j$ 次序(冲突)串行化。

# 基于有效性检查的协议

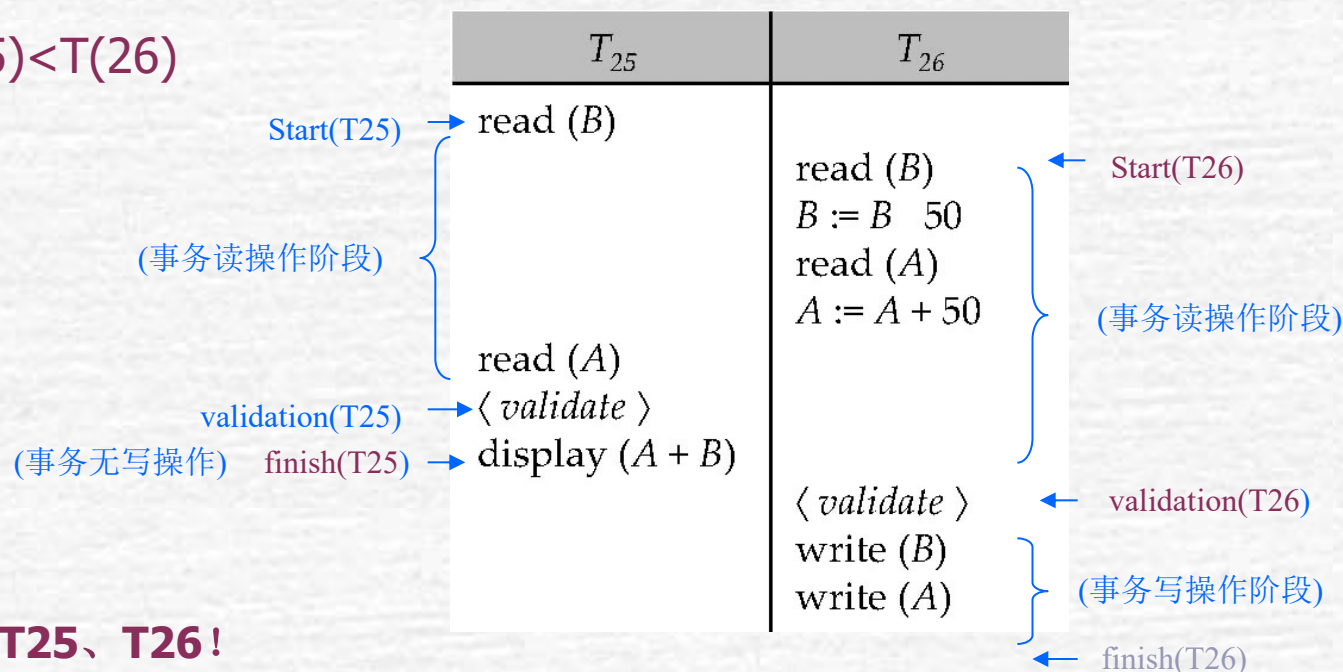
3. 什么是悲观的并发控制？乐观的并发控制？

该调度的事务有效性检查结果如何？

## Schedule Produced by Validation

- Example of schedule produced using validation

显然:  $T(25) < T(26)$



可串行化为 **T25、T26** !

不满足第一条条件! 1)  $finish(T_{25}) < start(T_{26})$  ?

但满足第二条条件: 2)  $start(T_{26}) < finish(T_{25}) < validation(T_{26})$ , and the set of data items written by

$T_{25}$  does not intersect(不相交) with the set of data items read by  $T_{26}$  ?



# 课堂小测试

- 进行时间戳排序协议的分析

$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read (Y)	read (Y)	write (Y)		read (X)
	read (Z)	write (Z)		read(W)
read (X)		write (W)	read (W)	write (Y)
				write (Z)

# 课程总结与作业安排

- 基本知识:
  - 时间戳排序协议
  - 快照隔离
  - 基于有效性检查的协议
- 扩展学习:
  - 如何设计一个时间戳排序协议算法?
- 作业

15章15. 28, 15. 29