# 第十五章：15：

# 并发控制Concurrency Control

数据库系统概念，第 7版。

# 基于锁的协议

- 锁是一种控制对数据项的并发访问的机制
- 数据项可以两种模式锁定：
  1. 独占（X）模式。数据项既可以读取也可以写入。使用lock-X指令请求X-lock 。
  2. 共享（S）模式。只能读取数据项。使用lock-S指令请求S-lock 。
- 向并发控制管理器发出锁定请求。交易可以只有在请求被批准后才能继续。

# 基于锁的协议 Lock-Based Protocols

- 锁兼容性矩阵

| | S | X |
|---|---|---|
| S | true | false |
| X | false | false |

- 如果请求的锁与其他事务已在该项目上持有的锁兼容,则该事务可能被授予对该项目的锁定
- 任意数量的事务都可以持有一个项目的共享锁,但如果任何事务持有该项目的独占锁,则没有其他事务可以持有该项目的任何锁

# 安排锁定授权 With Lock Grants

- 从现在起，不再明确说明赠款
  - 假设授权发生在锁定请求之后的下一条指令之前
- 锁定协议是所有事务在请求和释放锁定时遵循的一组规则
- 锁定协议通过限制可能的调度集来强制执行可串行化

| $T_1$ | $T_2$ | concurrency-control manager |
|---|---|---|
| lock-X($B$) | | |
| | | grant-X($B$, $T_1$) |
| read($B$) | | |
| $B := B - 50$ | | |
| write($B$) | | |
| unlock($B$) | | |
| | lock-S($A$) | |
| | | grant-S($A$, $T_2$) |
| | read($A$) | |
| | unlock($A$) | |
| | lock-S($B$) | |
| | | grant-S($B$, $T_2$) |
| | read($B$) | |
| | unlock($B$) | |
| | display($A + B$) | |
| lock-X($A$) | | |
| | | grant-X($A$, $T_1$) |
| read($A$) | | |
| $A := A + 50$ | | |
| write($A$) | | |
| unlock($A$) | | |

# 僵局 Deadlock

- 考虑部分时间表

| $T_3$ | $T_4$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | lock-S($B$) |
| lock-X($A$) | |

- T3和T4都无法取得进展　执行lock-S(B)导致T4等待T3释放其对B 的锁定,而执行 lock-X(A)导致
  T3等待T4释放其对A的锁定 ▪这种情况是死锁

情况

- 要处理死锁,必须回滚T3或T4之一
  并且它的锁被释放

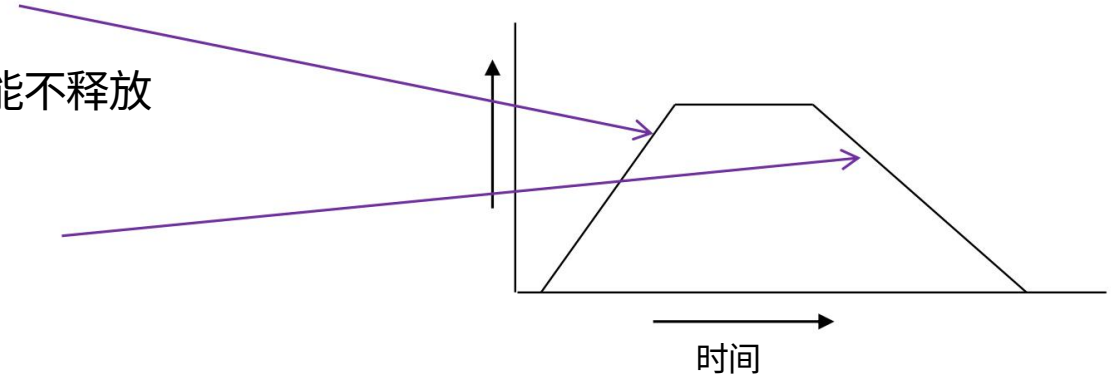
# 僵局 Deadlock

- 大多数锁定协议都存在死锁的可能性
- 当一个事务无限期地无法完成其任务而其他事务继续进行时,就会发生饥饿
  - 一个事务可能正在等待一个项目的 X 锁,而一个其他事务请求的序列并被授予对同一项目的 S 锁
  - 由于死锁,同一事务被反复回滚
- 可以设计并发控制管理器来防止饥饿

# 两阶段锁定协议 Two-Phase Locking Protocol

- 确保冲突可序列化调度的协议

- 第一阶段：成长阶段
  - 事务可能获得锁
  - 事务可能不释放锁
- 第二阶段：收缩阶段
  - 事务可能会释放锁
  - 事务可能无法获得锁




- 该协议确保可串行化。可以证明交易可以按照其锁点的顺序进行序列化

（即事务获得其最终锁定的点）

# 两阶段锁定协议 Two-Phase Locking Protocol

- 带锁转换的两阶段锁定协议：
  - 成长阶段： ·可以
    在项目上获得 lock-S
    - ·可以在项目上获得 lock-X
    - ·可以将lock-S 转换为 lock-X （升级）
  - 收缩阶段：
    - ·可以释放一个锁-S
    - ·可以释放一个锁-X
    - ·可以将 lock-X 转换为 lock-S （降级）
- 该协议确保可串行化

# 两阶段锁定协议 Two-Phase Locking Protocol

- 需要扩展基本的两相锁定，以确保从级联回滚中恢复自由

  - 严格的两阶段锁定：一个事务必须持有它所有的独占锁定直到它提交/中止
    - 确保可恢复性并避免级联回滚（避免脏读　在提交之前没有人可以读取更新的数据项）

  - 严格的两阶段锁：一个事务必须持有所有锁直到提交/中止
    - 避免不可重复读取（当一个共享锁被持有）
    - 事务可以按照提交的顺序进行序列化

- 大多数数据库实施严格的两阶段锁定

# 自动获取锁 Automatic Acquisition of Locks

- 事务Ti发出标准的读/写指令,没有明确的
  锁定调用
- 操作read(D)被处理为:

```
如果Ti锁定了D
  然后
      读（D)否
  则开始
      如有必要,等到没有其他
        事务在D上有一个lock-X
      授予Ti在D 上的lock-S ;读
      (D)结束
```

# 自动获取锁 Automatic Acquisition of Locks

- 操作write(D)被处理为：

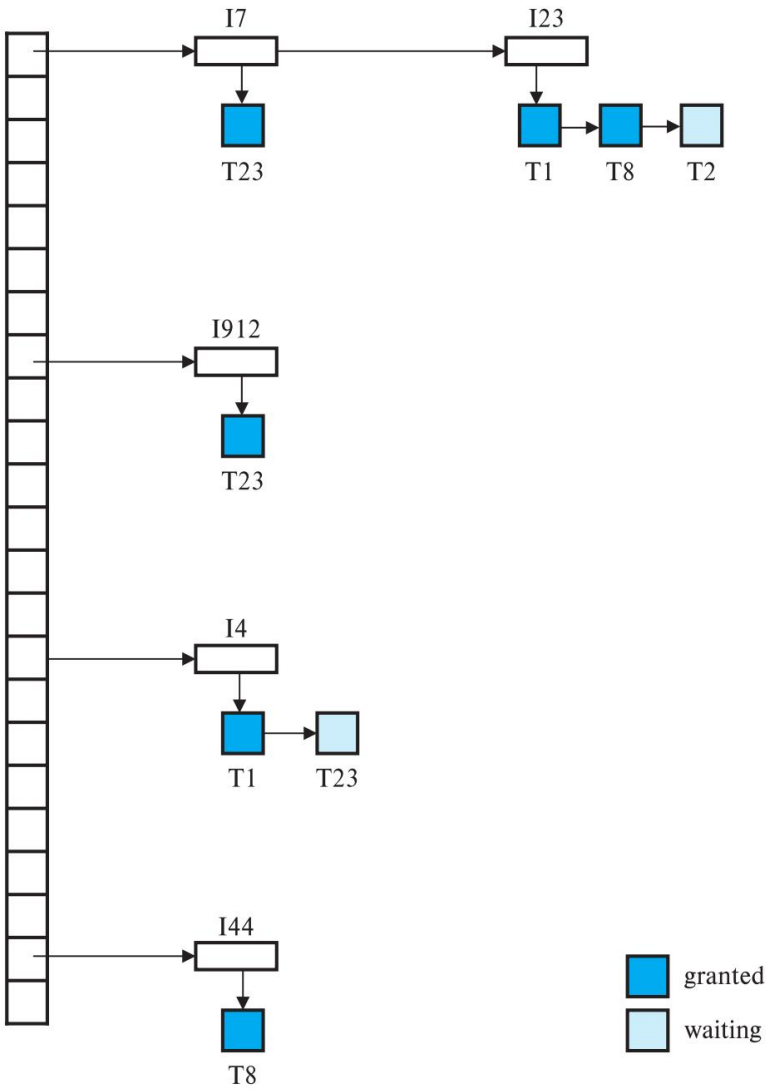
  如果Ti在D上有一个lock-X然
  后write(D) else 开始,如
  果需要,等到没有其他事
  务在 D 上有任何锁,如果Ti
  在D上有一个lock-S然后将D上的锁升级到lock-X else grant Ti a lock-X
  on D write(D) end;

- 在提交或中止后释放所有锁

# 锁定的实现

- 锁定管理器可以作为一个单独的进程来实现
- 事务可以将锁定和解锁请求作为消息发送
- 锁管理器通过发送锁授权来回复锁请求消息（或要求事务回滚的消息，以防死锁）
  - 请求事务等待直到其请求得到答复
- 锁管理器维护一个称为锁表的内存数据结构，以记录已授予的锁和待处理的请求
  - 使用哈希表，对数据项的名称进行哈希处理，以查找数据项的链表

# 锁表




- 数据项以 I 为前缀,事务以 T 为前缀
- 较深的矩形表示已授予锁,较浅的矩形表示等待请求
- 锁表还记录了锁的类型授予或请求
- 新请求被添加到数据项请求队列的末尾,如果它与所有早期锁兼容,则授予它
- 解锁请求会导致请求被删除,并检查以后的请求是否现在可以被授予
- 如果事务中止,则删除该事务的所有等待或授予的请求

# 死锁处理

- 如果有一组事务,使得集合中的每个事务都在等待集合中的另一个事务,系统就会死锁

| $T_3$ | $T_4$ |
|---|---|
| lock-X($B$) | |
| read($B$) | |
| $B := B - 50$ | |
| write($B$) | |
| | lock-S($A$) |
| | read($A$) |
| | lock-S($B$) |
| lock-X($A$) | |

# 死锁预防 Deadlock Prevention

- 等待死亡方案　非抢占式
  - 较旧的事务可能会等待较年轻的事务释放数据项
  - 较新的事务从不等待较旧的事务;越年轻
    事务被回滚
  - 一个事务在获得锁之前可能会死掉几次
  - 例如,假设事务 T14、T15 和 T16 具有
    时间戳分别为 5、10 和 15。如果 T14 请求 T15 持有的数据项,则 T14 将等待。如果 T16 请求 T15 持有的数据项,则 T16 将回滚。

# 死锁预防 Deadlock Prevention

- 伤口等待计划 先发制人
  - 较年轻事务的较早事务伤口（强制回滚）
    而不是等待它
  - 较新的事务可能会等待较旧的事务
  - 对于上面的事务T14、T15和T16，其时间戳分别为5、10和15，如果T14请求T15持有的数据项，则该数据项将从T15抢占，T15将回滚。如果 T16 请求 T15 持有的数据项，则 T16 将等待

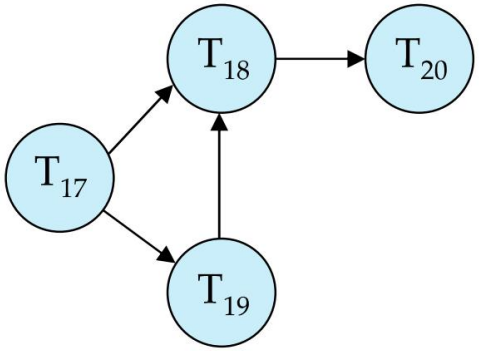
- 在这两种方案中，回滚的事务都以其原始事务重新启动时间戳
  - 确保旧事务优先于新事务，
    避免饥饿

# 死锁预防 Deadlock Prevention

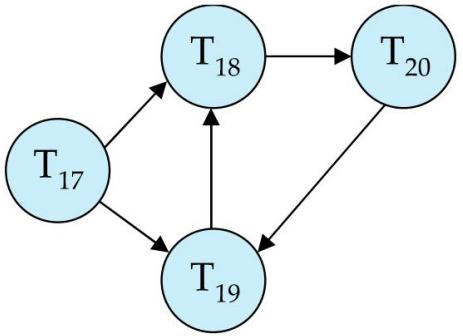
- 基于超时的方案
  - 事务仅在指定的时间内等待锁定。之后，等待超时，事务回滚
  - 确保死锁在发生时通过超时解决
  - 易于实施
  - 但在没有死锁的情况下可能会不必要地回滚事务
  - 难以确定超时间隔的正确值
  - 饥饿也是可能的

# 死锁检测 Deadlock Detection

- 等待图
  - 顶点:交易
  - 来自Ti →Tj的边缘。 :如果Ti正在等待以冲突模式持有的锁 Tj
- 系统处于死锁状态当且仅当等待图有一个循环
- 定期调用死锁检测算法来查找循环




没有循环的等待图




带循环的等待图

# 死锁恢复 k Recovery

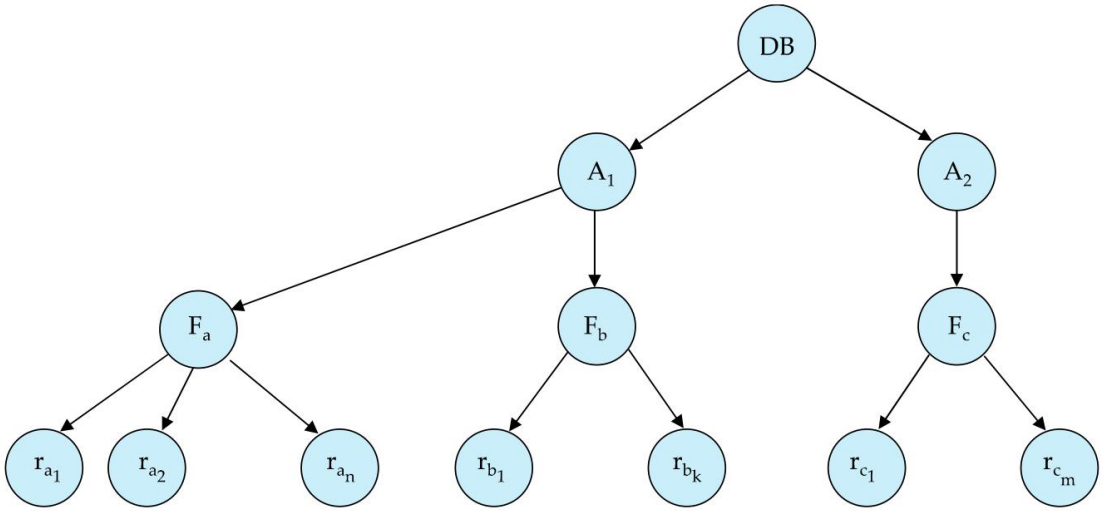
- 检测到死锁时
  - 某些事务将不得不回滚（成为牺牲品）以打破死锁循环
    - 选择该交易作为将产生最低成本的受害者
  - 回滚 确定回滚事务的距离
    - 总回滚：中止事务然后重新启动它
    - 部分回滚：仅将受害事务回滚至
      需要释放循环中另一个事务正在等待的锁
      - 死锁检测机制应该决定哪个锁
        选择的事务需要释放才能打破死锁。所选事务必须回滚到它获得第一个锁的点，撤消它在该点之后执行的所有操作
- 饥饿可能发生
  - 始终选择同一事务作为受害者
  - 一种解决方案：死锁集中最旧的事务永远不会被选为
    受害者

# 多粒度 Multiple Granularity

- 允许数据项具有各种大小并定义数据的层次结构
  粒度,其中小粒度嵌套在较大粒度中
- 可以用图形表示为一棵树
- 当一个事务显式锁定树中的一个节点时,它会隐式锁定所有
  节点的后代处于相同模式
- 锁定的粒度（在树中完成锁定的级别）：
  - 细粒度（树中较低）:高并发、高锁定
    高架
  - 粗粒度（树中较高）:锁定开销低、低
    并发

# 粒度层次结构示例 Granularity Hierarchy

- 从最粗（最高）级别开始的级别是
  - 数据库
  - 面积
  - 文件
  - 记录



# 意向锁定模式 Intent Lock Modes

- 假设一个事务希望锁定整个数据库
  - 为此,它必须锁定层次结构的根
  - 系统如何确定根节点是否可以锁定:
    - 一种可能性是搜索整棵树,这距离远
      高效的
- 获得这些知识的一种更有效的方法是引入一类新的
  锁定模式,称为意图锁定模式
  - 如果节点以意向模式锁定,则显式锁定在树的较低级别(即以更精细的粒度)完成。
- 在显式锁定该节点之前,对该节点的所有祖先都设置了意向锁
  - 因此,事务不需要搜索整个树来确定它是否可以成功锁定节点
- 希望锁定节点 Q 的事务必须遍历树中的路径
  从根到 Q
  - 在遍历树时,事务以意向模式锁定各个节点

# 意向锁定模式Lock Modes

- 除了 S 和 X 锁定模式之外,还有另外三种锁定模式
  多粒度:
  - intent-shared (IS):表示将在某些后代节点上请求共享锁
  - intent-exclusive (IX):表示排他锁将被
    在某些后代节点上请求
  - shared and intent-exclusive (SIX):当前节点被锁定在共享模式,但在某些后代节点上会请求排他锁
- 意向锁允许将更高级别的节点锁定在 S 或 X 模式,而无需检查所有后代节点

# 意向锁定模式的兼容性矩阵 Compatibility Matrix for Intention Lock Modes

- 所有锁定模式的兼容性矩阵为：

| | IS | IX | S | SIX | X |
|---|---|---|---|---|---|
| IS | true | true | true | true | false |
| IX | true | true | false | false | false |
| S | true | false | true | false | false |
| SIX | true | false | false | false | false |
| X | false | false | false | false | false |

# 多粒度锁定方案 Multiple Granularity Locking Scheme

▪事务Ti可以锁定一个节点Q,使用以下规则： 1.必须遵守锁定兼容性矩阵2.必须先锁定树的根,并且可以在任意位置锁定

模式

3.只有当Q的父节点当前在 IX 或 IS 模式下被Ti锁定时,节点Q才能在 S 或 IS 模式下被Ti锁定4.在 X、SIX 或 IX 模式下,节点Q可以被Ti锁定仅当Q的父节点当前在 IX 或 SIX 模式下被Ti锁定5. Ti只有在之前没有解锁任何节点时才能锁定节点（即Ti是两阶段的）

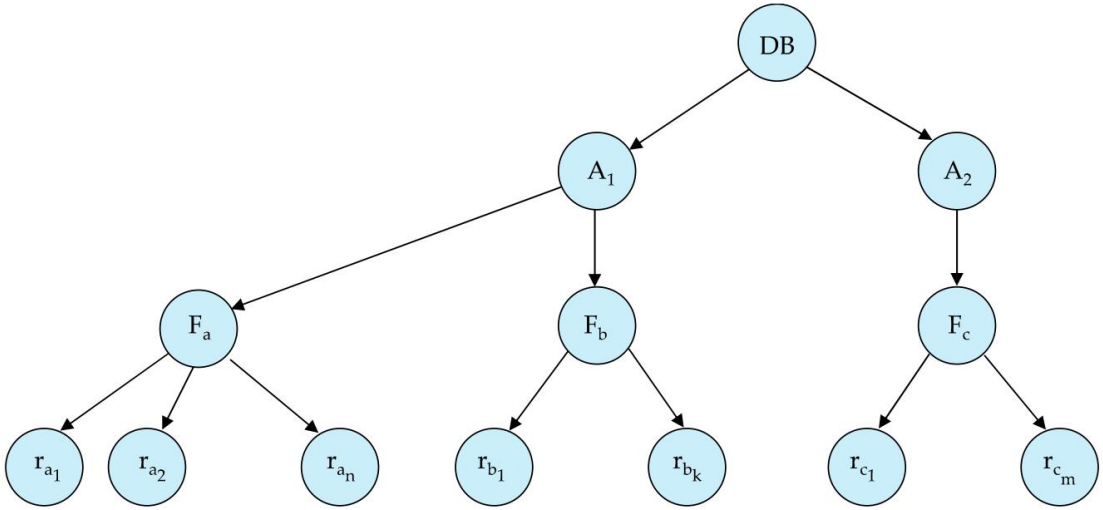
6.只有当 Q 的子节点当前都不存在时， Ti才能解锁节点Q

被Ti锁定 ▪观察

锁是按照从根到叶的顺序获取的,而它们是按照从叶到根的顺序释放的

# 多粒度锁定方案 Multiple Granularity Locking Scheme

- 假设事务T21读取文件Fa中的记录ra2 。然后， T21需要将数据库、 A1区、 Fa锁在IS模式（并按顺序），最后将ra2锁在S模式
- 假设事务T22修改了文件Fa中的记录ra9 。然后， T22需要在IX模式下锁定数据库、 A1区和文件Fa （并按顺序），最后在X模式下锁定ra9
- 假设事务T23读取文件Fa 中的所有记录。然后， T23需要锁定数据库和区域A1 （并按此顺序)处于IS模式,最后将Fa锁定为 S 模式
- 假设事务T24读取整个数据库。锁定数据库后可以这样做
  在 S 模式下
- 请注意,事务T21、 T23和T24可以同时访问数据库。交易T22可以与T21并发执行,但不与T23或T24一起执行



# 基于时间戳的协议

- 每个事务Ti在进入系统
  - 每笔交易都有唯一的时间戳
  - 较新事务的时间戳严格大于以前的事务那些
  - 时间戳可以基于逻辑计数器
    - 实时可能不是唯一的
    - 可以使用逻辑计数器来保证唯一性
- 基于时间戳的协议管理并发执行，使得时间戳顺序 = 可序列化顺序

# 时间戳排序协议 Timestamp Ordering Protocol

时间戳排序( TSO) 协议

- 为每个数据Q维护两个时间戳值：
  - W-timestamp(Q)是成功执行write(Q)的任何事务的最大时间戳
  - R-timestamp(Q)是成功执行read(Q)的任何事务的最大时间戳
- 对读写操作施加规则，以确保
  - 任何冲突的操作都按时间戳顺序执行
  - 乱序操作导致事务回滚

# 时间戳排序协议 Timestamp Ordering Protocol

- 假设一个事务Ti发出一个read(Q)

1. 如果 TS(Ti ) W-timestamp(Q),则Ti尝试读取Q的未来值 - 因此,读取操作被拒绝,并且Ti被回滚
2. 如果 TS(Ti ) W-timestamp(Q),则执行读操作,并且 R-timestamp(Q)设置为

   最大值（R-时间戳（Q）， TS（Ti））

# 时间戳排序协议 Timestamp Ordering Protocol

- 假设事务Ti发出write(Q)

1. 如果 TS(Ti ) < R-timestamp(Q),那么Ti正在产生的Q的值是之前需要的,并且系统假设该值永远不会产生因此,写操作被拒绝,并且Ti被回滚。

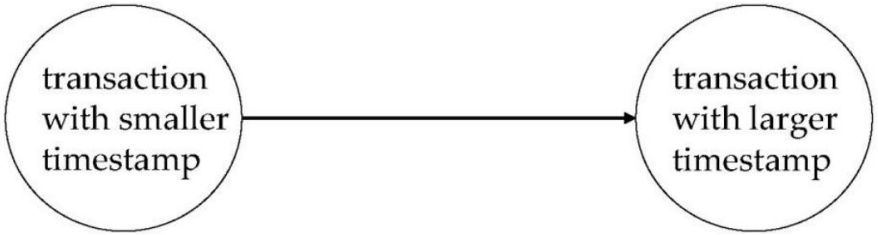
2. 如果 TS(Ti ) < W-timestamp(Q),那么Ti正试图写入（产生)一个过时（被取代)的Q值因此,这个写入操作被拒绝,并且Ti被回滚3.否则,写操作被执行,W-timestamp(Q)为

设置为 TS (Ti )

# 时间戳排序协议的正确性



- 时间戳排序协议保证了可串行性,因为优先图中的所有弧都具有以下形式:




因此,优先图中将没有循环

- Timestamp 协议确保免于死锁,因为没有事务
  永远等待。

# TSO 下的时间表示例

假设最初：

R-TS(A) = W-TS(A) = 0

R-TS(B) = W-TS(B) = 0

假设 TS(T25) = 25 并且

TS(T26) = 26

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

- TSO 下的top schedule 是有效的，因为T25只进行Reads，它不读取T26为同一日期项写入的任何未来值

- T27需要回滚，因为它试图在其 write(Q) 中为 Q 生成一个过时的值

认为：

R-TS(Q)=W-TS(Q)=0

TS(T27) = 27

TS(T28) = 28

| $T_{27}$ | $T_{28}$ |
|---|---|
| read($Q$) | |
| | write($Q$) |
| write($Q$) | |

# 托马斯的写规则

- 时间戳排序协议的修改版本，在某些情况下可以忽略过时的写入操作
- 当Ti试图写入数据项Q 时，如果 TS(Ti ) < W-timestamp(Q)，那么Ti正试图产生一个过时的值 {Q}
  - 而不是像时间戳排序协议那样回滚Ti
    已经做了，这个写操作可以忽略
- 否则此协议与时间戳排序协议相同。
- Thomas 的写入规则允许更大的潜在并发性