# CSCD43: Database Systems Technology

## Lecture 14

*Wael Aboulsaadat*

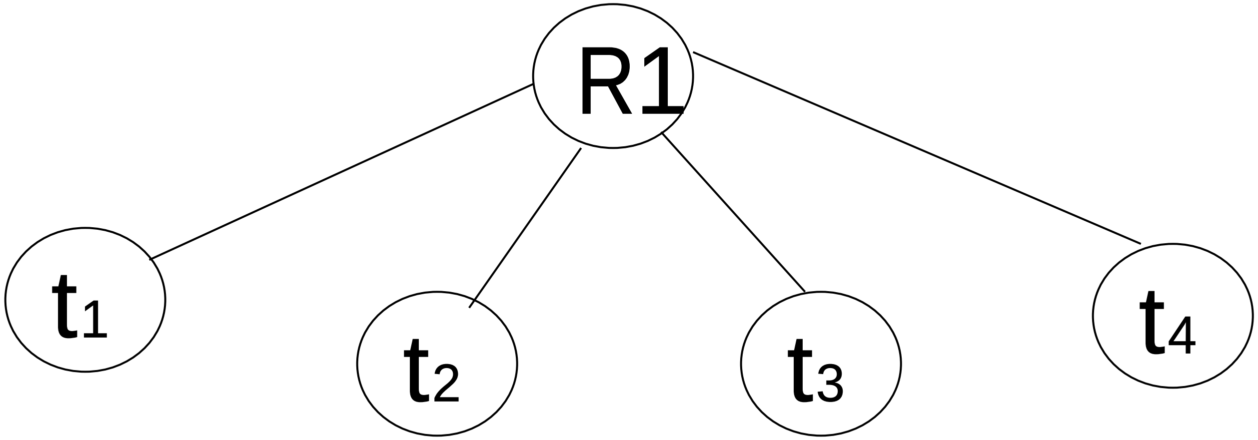
Acknowledgment: these slides are based on Prof. Garcia-Molina & Prof. Ullman slides accompanying the course's textbook.

# Concurrency Control

# How to enable concurrent access?

- Locking
- Timestamps

# Locking for B+ Tree

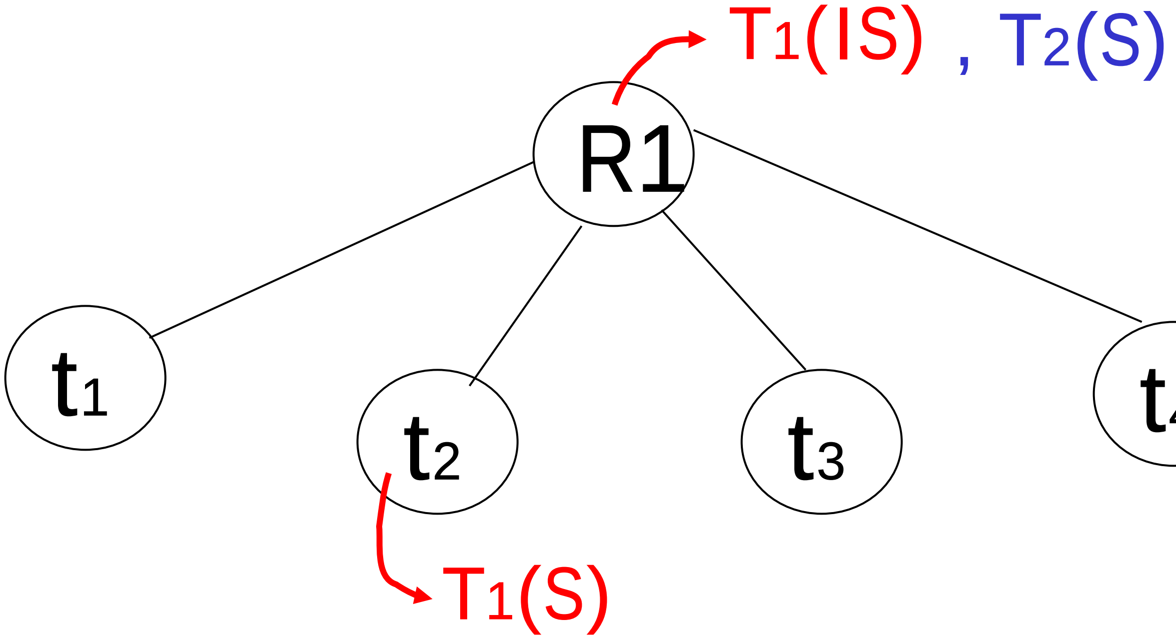


# Warning Locks

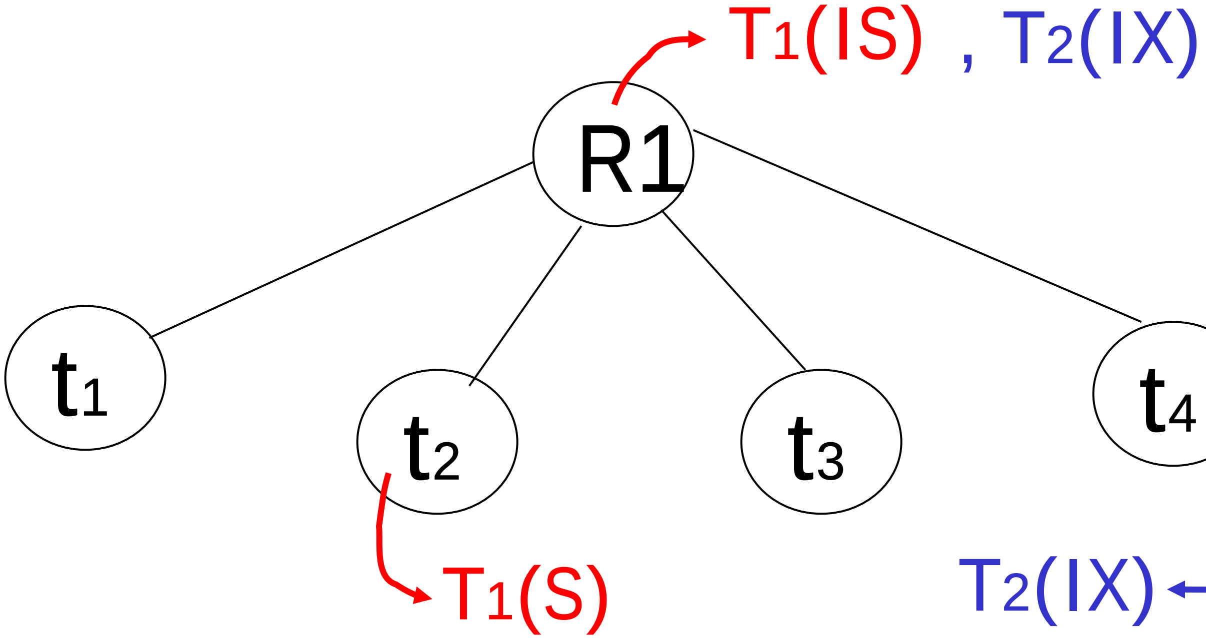
- Denoted by an I (for intention)

- Rules:
  - Always begin at the root of the tree.
  - If we are at the right element, request S or X lock
  - Else, request a warning lock IS or IX. If granted, move down the tree else wait.

# Example



# Example



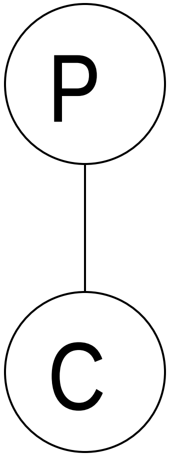
# Multiple granularity

| Comp | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| | IS | | | | |
| Holder | IX | | | | |
| | S | | | | |
| | X | | | | |

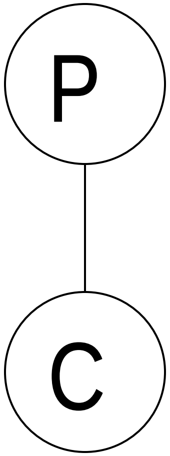
# Multiple granularity

| Comp | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| Holder | IS | T | T | T | F |
| | IX | T | T | F | F |
| | S | T | F | T | F |
| | X | F | F | F | F |

| Parent locked in | Child can be locked in |
|---|---|
| IS | |
| IX | |
| S | |
| X | |



| Parent locked in | Child can be locked in |
|---|---|
| IS | IS, S |
| IX | IS, S, IX, X |
| S | S, IS |
| X | none |

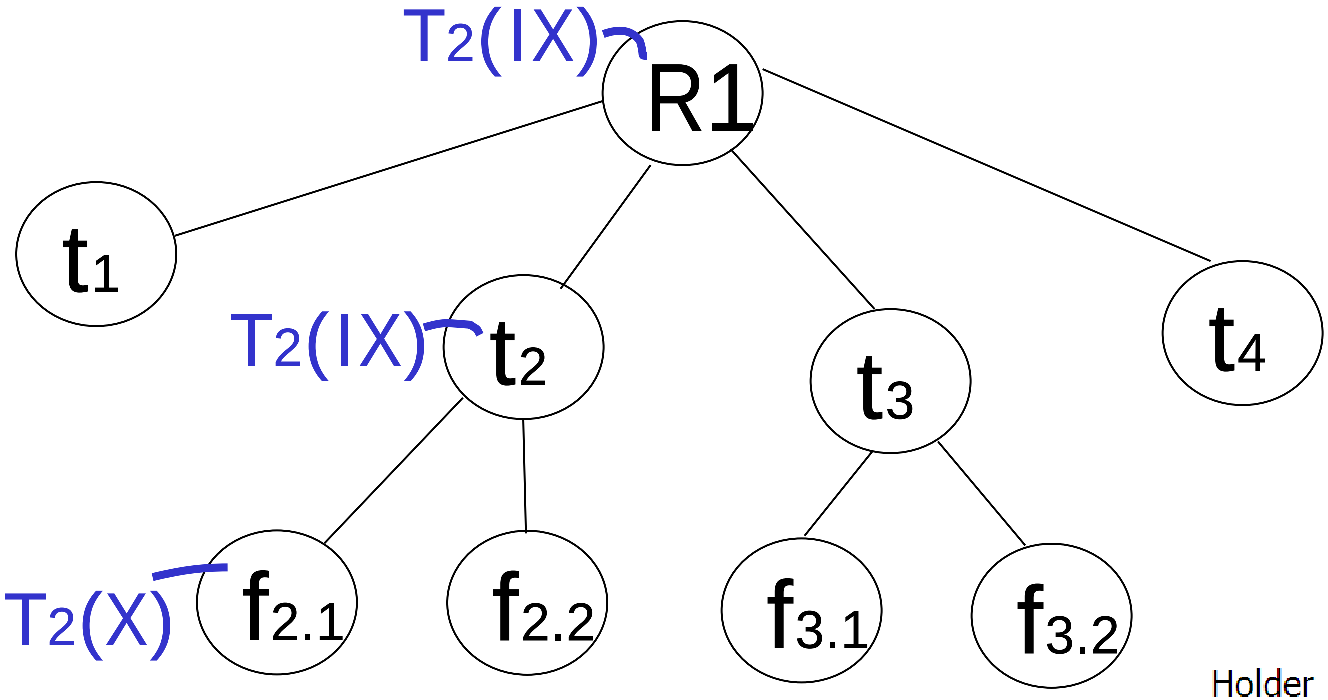


# Rules

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) can be locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X, IX only if parent(Q) locked by Ti in IX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q’s children are locked by Ti

# Exercise:

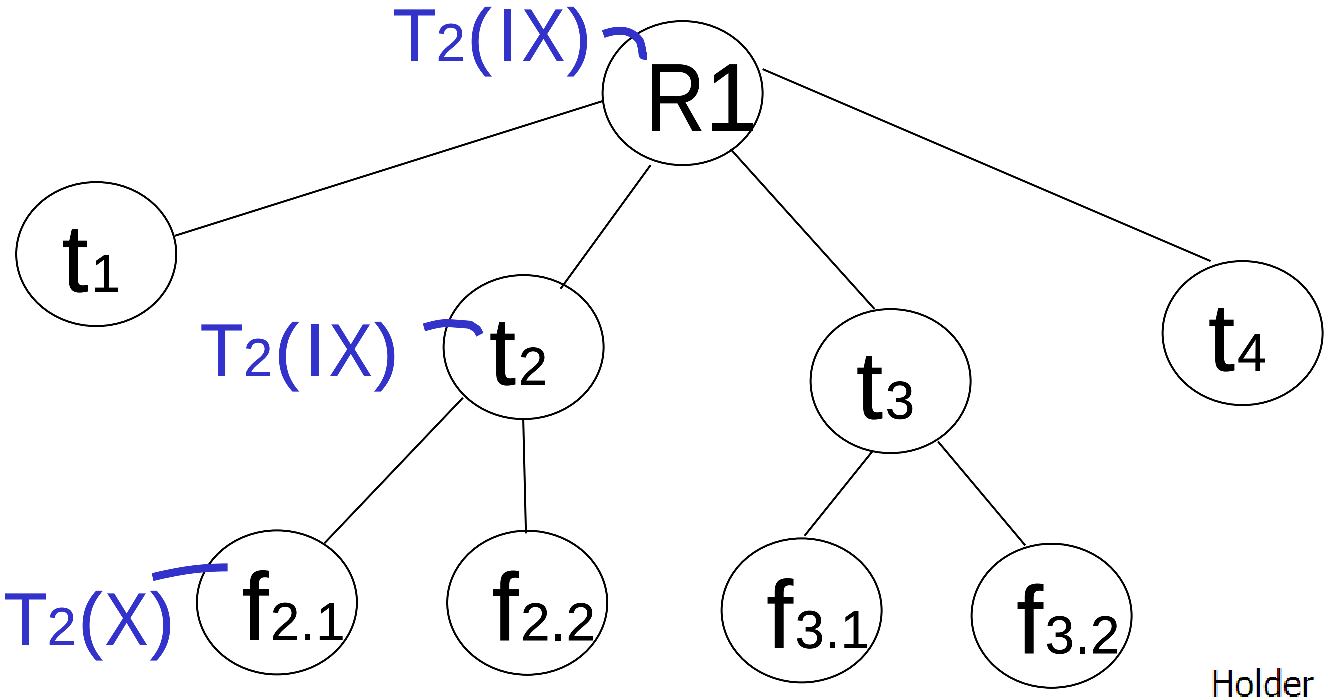
- Can $T_2$ access object $f_{2.1}$ in X mode? What locks will $T_2$ get?



| | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| Holder | IS | T | T | T | F |
| | IX | T | T | F | F |
| | S | T | F | T | F |
| | X | F | F | F | F |

# Exercise:

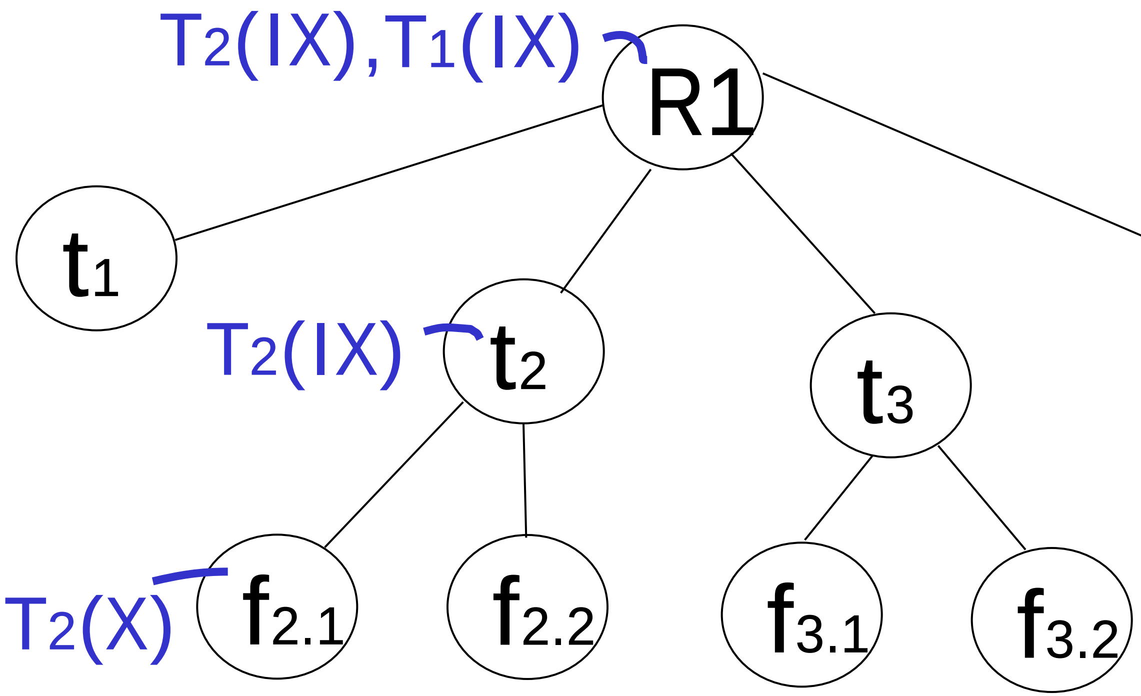
- Can $T_1$ access object $f_{2.2}$ in X mode? What locks will $T_1$ get?

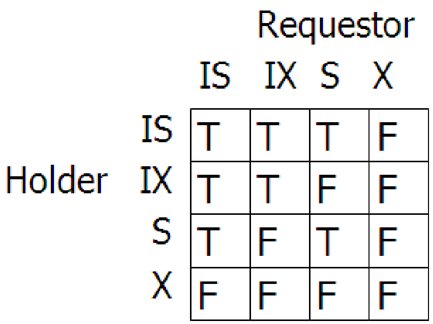


| | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| Holder | IS | T | T | T | F |
| | IX | T | T | F | F |
| | S | T | F | T | F |
| | X | F | F | F | F |

# Exercise:

- Can $T_1$ access object $f_{2.2}$ in X mode? What locks will $T_1$ get?

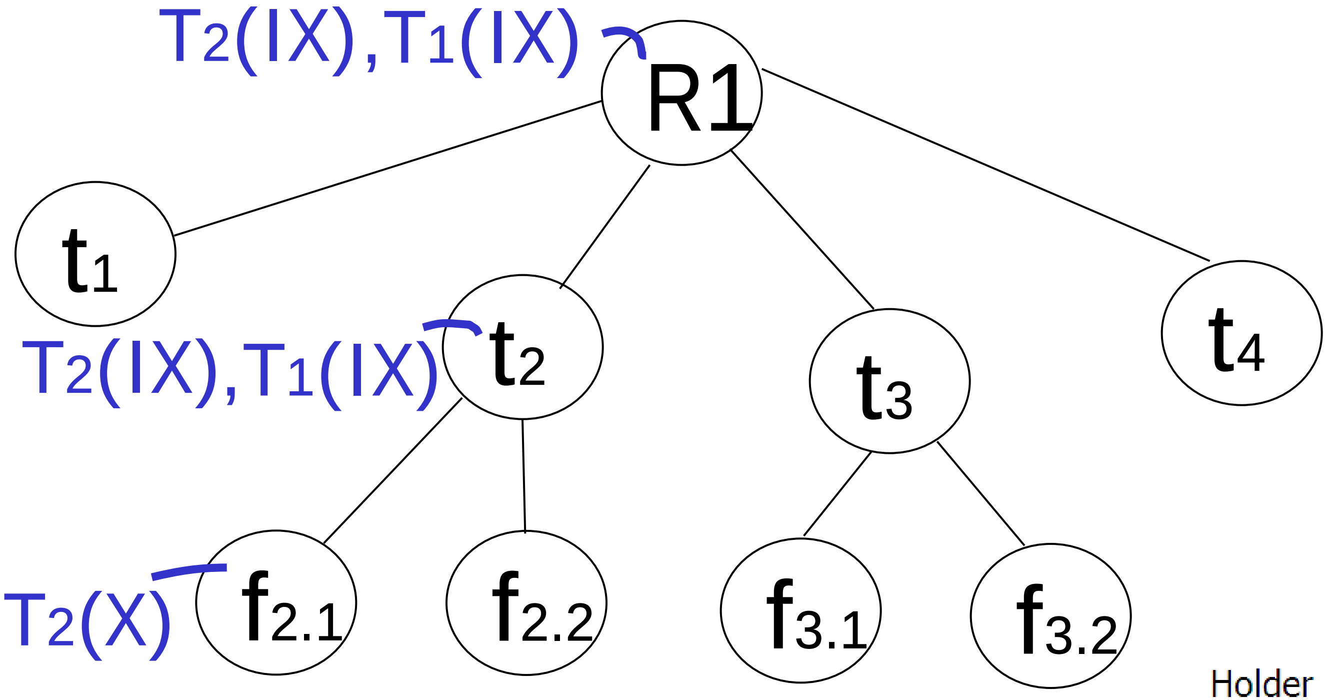




| | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| Holder | IS | T | T | T | F |
| | IX | T | T | F | F |
| | S | T | F | T | F |
| | X | F | F | F | F |

# Exercise:

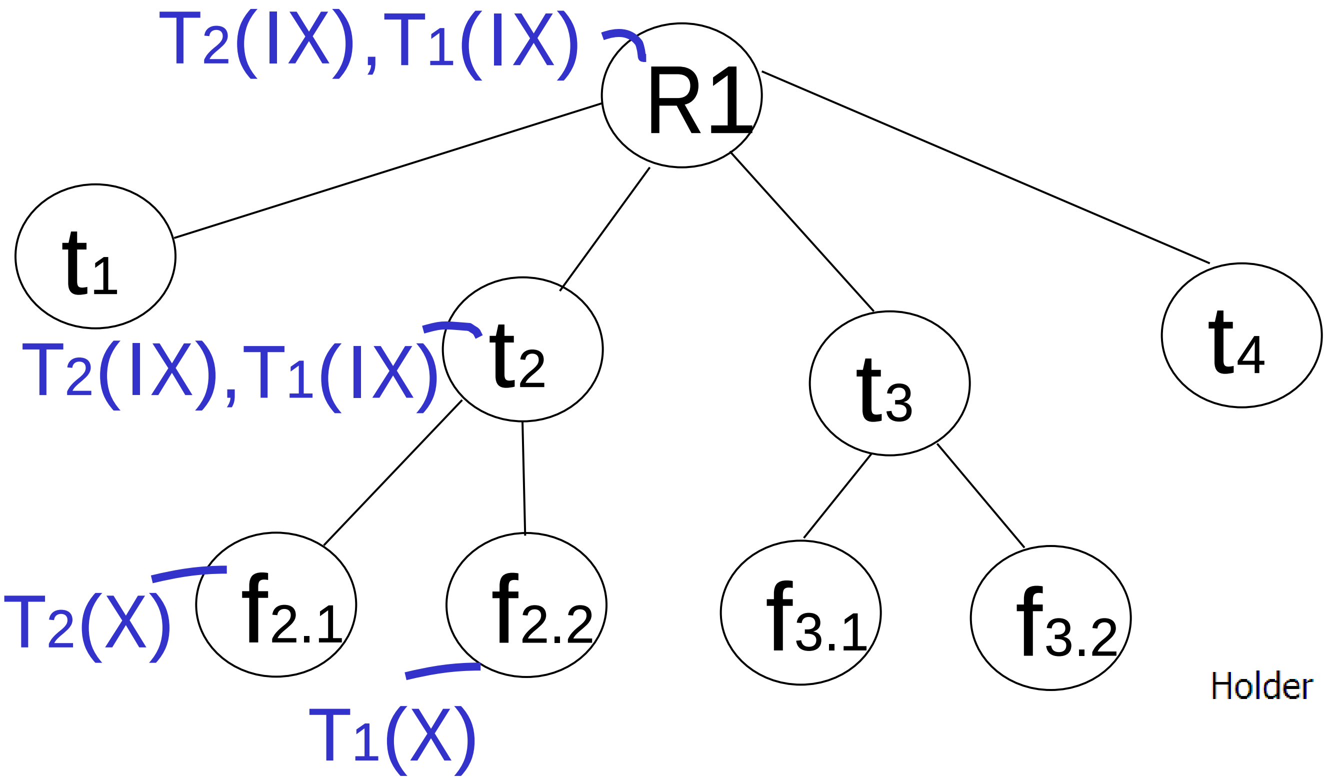
- Can $T_1$ access object $f_{2.2}$ in X mode? What locks will $T_2$ get?



| | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| Holder | IS | T | T | T | F |
| | IX | T | T | F | F |
| | S | T | F | T | F |
| | X | F | F | F | F |

# Exercise:

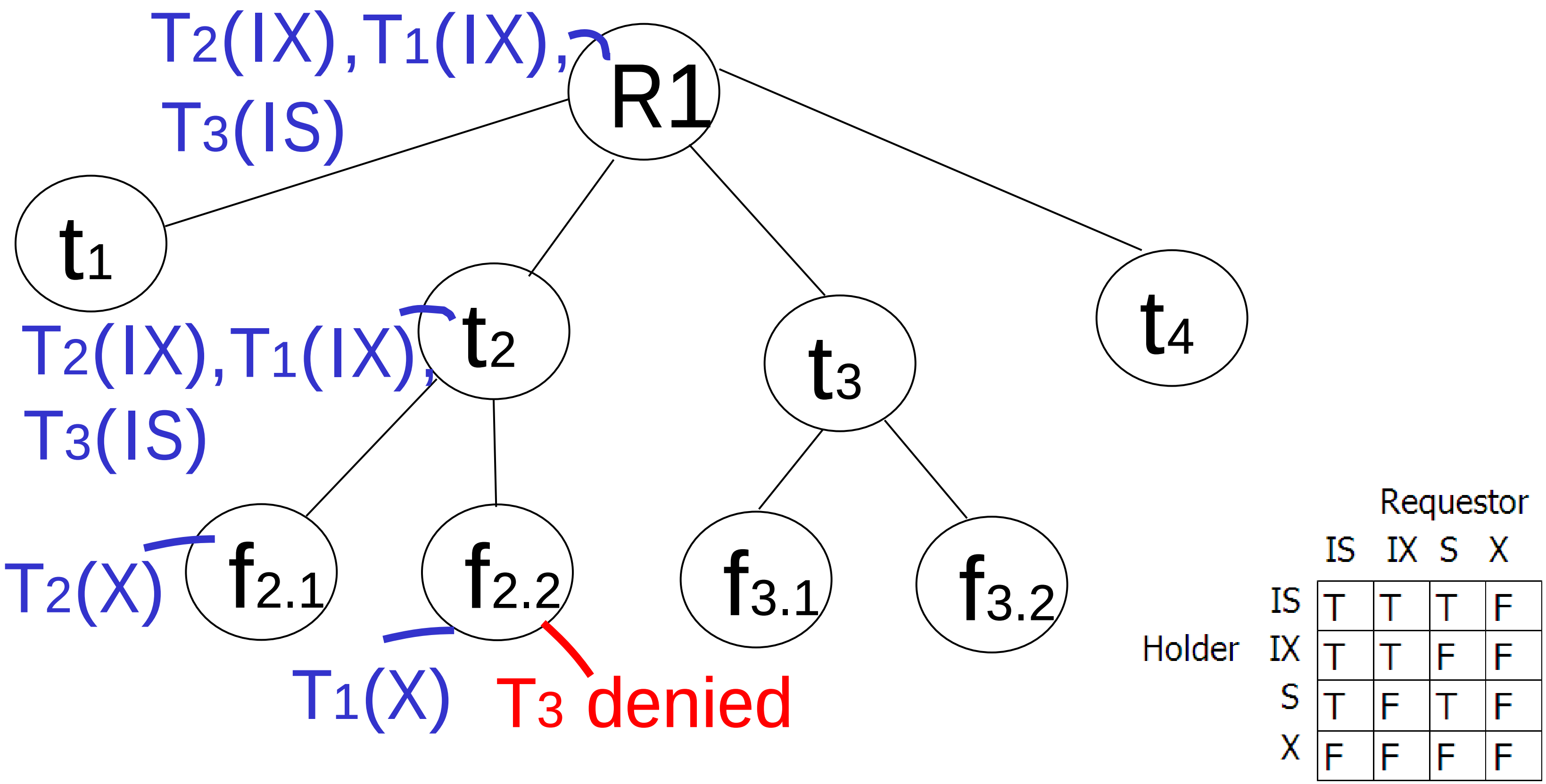
- Can $T_1$ access object $f_{2.2}$ in X mode? What locks will $T_1$ get?



| | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| | IS | T | T | T | F |
| Holder | IX | T | T | F | F |
| | S | T | F | T | F |
| | X | F | F | F | F |

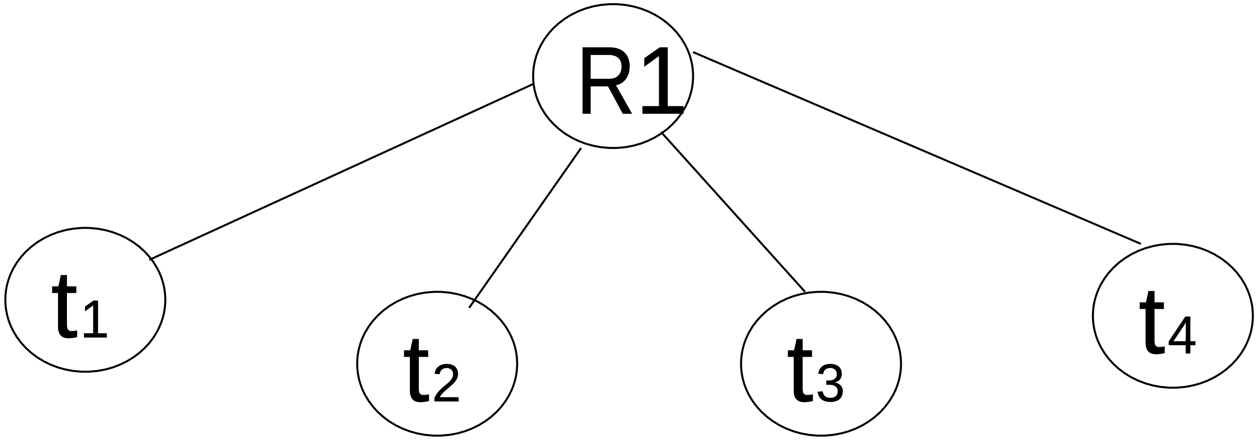
# Exercise:

- Can $T_3$ access object $f_{2.2}$ in S mode? What locks will $T_3$ get?



| | | Requestor | | | |
|---|---|---|---|---|---|
| | | IS | IX | S | X |
| Holder | IS | T | T | T | F |
| | IX | T | T | F | F |
| | S | T | F | T | F |
| | X | F | F | F | F |

# How to handle Insert and delete?



# Modifications to locking rules:

(1)Get exclusive lock on A before deleting A

(2) At insert A operation by Ti, Ti is given exclusive lock on A

# Still have a problem: **Phantoms**

Example: relation R (E#,name,...)

constraint: E# is key

use tuple locking

| R | E# | Name | .... |
|---|---|---|---|
| o1 | 55 | Smith | |
| o2 | 75 | Jones | |

$T_1$: Insert <99,Gore,...> into R

$T_2$: Insert <99,Bush,...> into R

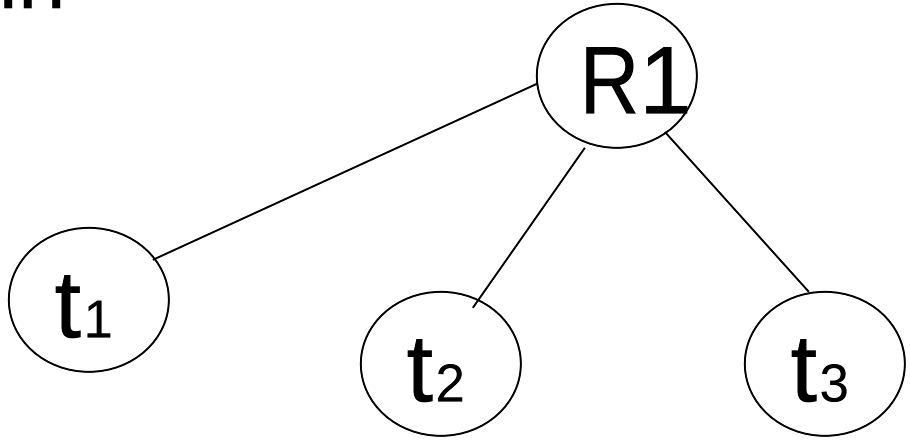
| $T_1$ | $T_2$ |
|---|---|
| $S_1(o_1)$ | $S_2(o_1)$ |
| $S_1(o_2)$ | $S_2(o_2)$ |
| Check Constraint | Check Constraint |
| ⋮ | ⋮ |
| Insert $o_3$[99,Gore,..] | |
| | Insert $o_4$[99,Bush,..] |

# Solution

- Use multiple granularity tree
- Before insert of node Q,

  lock parent(Q) in

  X mode



# Back to example

$T_1$: Insert<99,Gore>

$T_2$: Insert<99,Bush>

| $T_1$ | $T_2$ |
|---|---|
| $X_1(R)$ | |
| | $X_2(R)$ ← *delayed* |
| Check constraint | |
| Insert<99,Gore> | |
| $U(R)$ | |
| | $X_2(R)$ |
| | Check constraint |
| | Oops! e# = 99 already in R! |

# Timestamps

- Assign a timestamp to each transaction
- Record timestamp of the transactions that last read and write each database element
- Use timestamp to enforce serializalibility

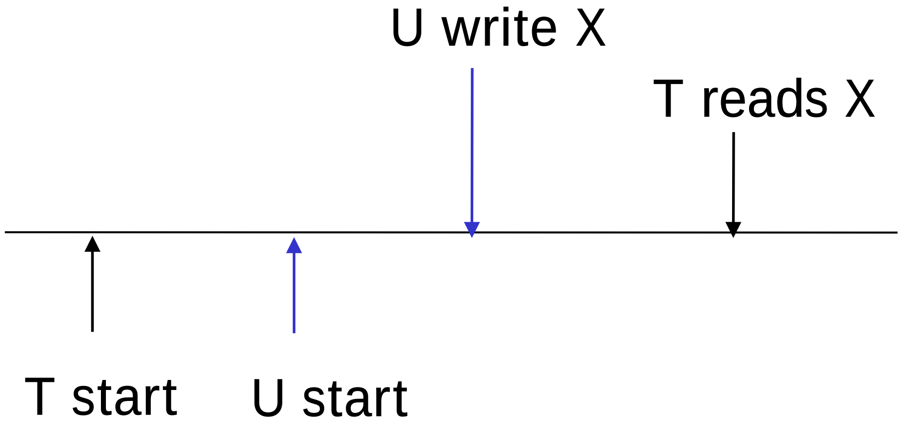
Optimistic approach:
fix things only if a violation occurs

# Timestamps

- RT(X): the latest read time of X
- WT(X): the latest write time of X
- C(X): the commit bit for X – true iff the most recent transaction to write X has already committed

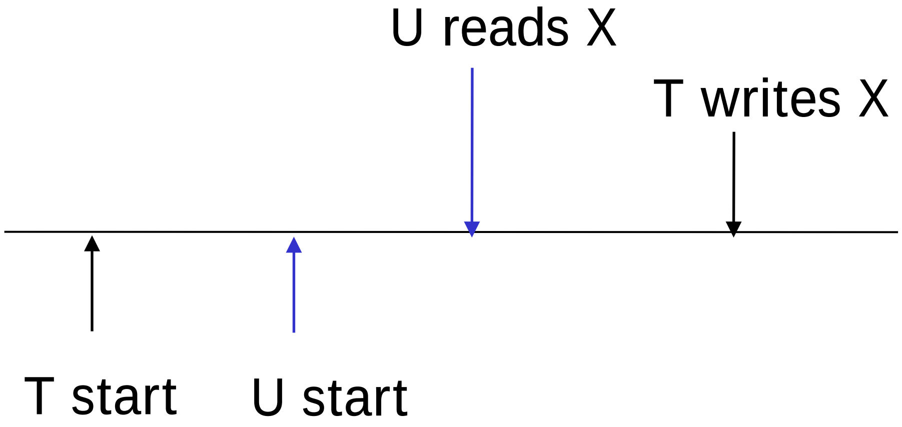
# Physically Unrealizable Behaviors

- Read too late



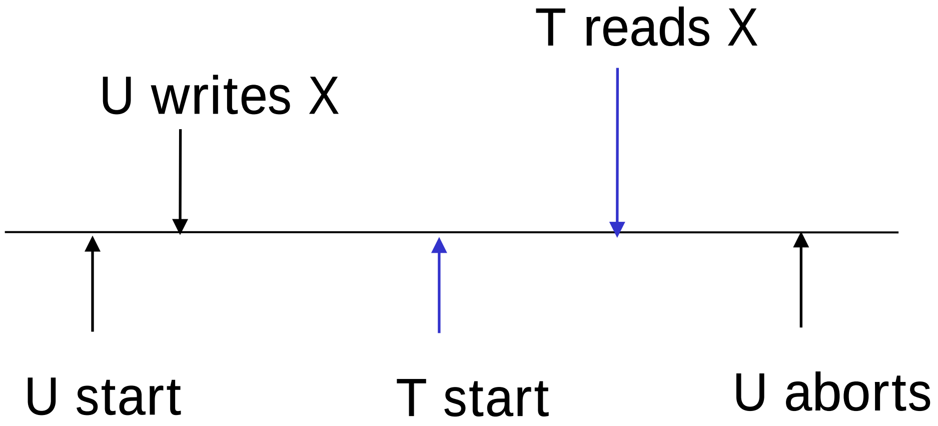
# Physically Unrealizable Behaviors

- Write too late



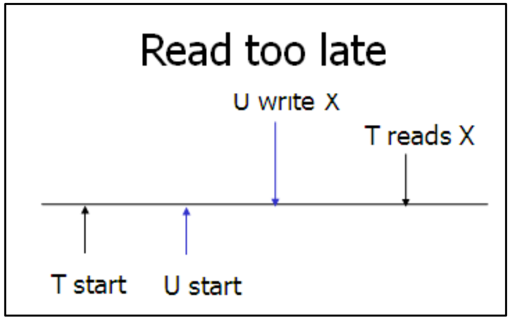
# Physically Unrealizable Behaviors

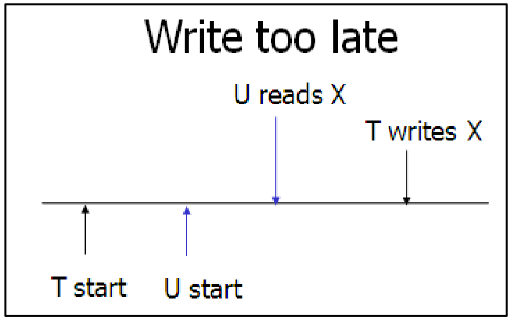
- Dirty Data Reading



# Three transactions under a timestamp-based scheduler

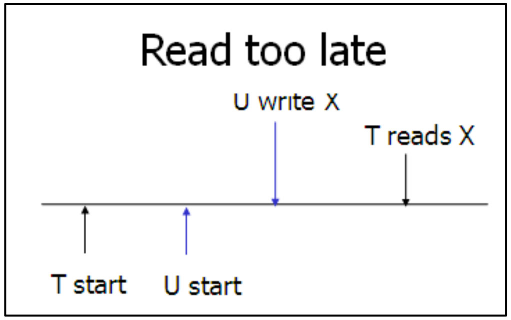
| *T1* | *T2* | *T3* | *A* | *B* | *C* |
|---|---|---|---|---|---|
| 200 | 150 | 175 | RT=0<br>WT=0 | RT=0<br>WT=0 | RT=0<br>WT=0 |
| r1(B); | | | | RT=200 | |

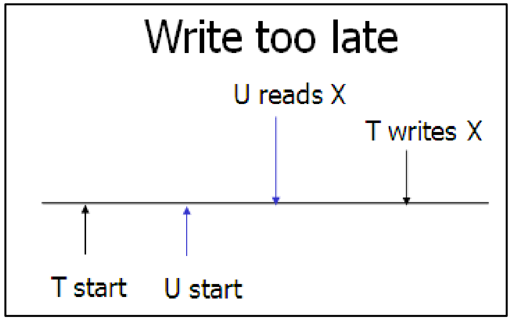




# Three transactions under a timestamp-based scheduler

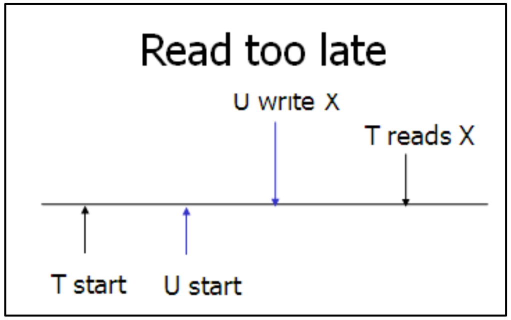
| **T1** | **T2** | **T3** | **A** | **B** | **C** |
|---|---|---|---|---|---|
| 200 | 150 | 175 | RT=0 | RT=0 | RT=0 |
| | | | WT=0 | WT=0 | WT=0 |
| r1(B); | | | | RT=200 | |
| | r2(A); | | RT=150 | | |
| | | r3(C); | | | RT=175 |
| w1(B); | | | | WT=200 | |

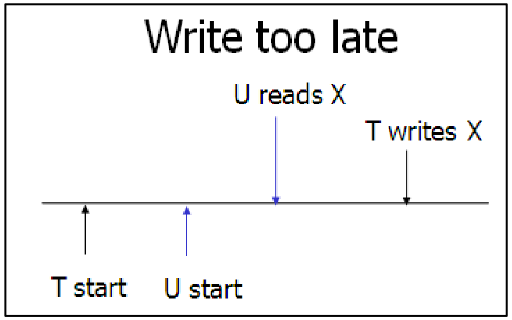




# Three transactions under a timestamp-based scheduler

| ***T1*** | ***T2*** | ***T3*** | ***A*** | ***B*** | ***C*** |
|---|---|---|---|---|---|
| 200 | 150 | 175 | RT=0 | RT=0 | RT=0 |
| | | | WT=0 | WT=0 | WT=0 |
| r1(B); | | | | RT=200 | |
| | r2(A); | | RT=150 | | |
| | | r3(C); | | | RT=175 |
| w1(B); | | | | WT=200 | |
| w1(A); | | | WT=200 | | |
| | w2(C); | | | | |
| | **Abort;** | | | | |

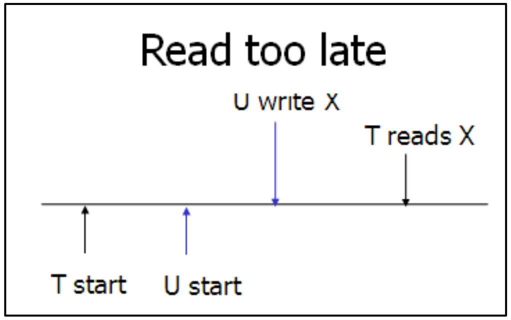


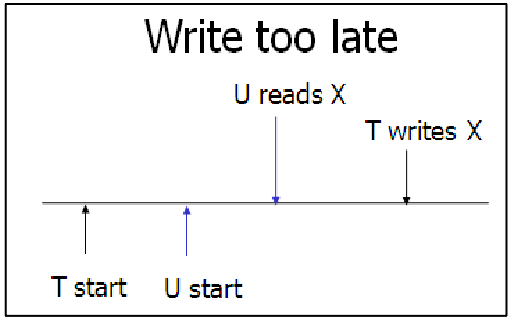


# Three transactions under a timestamp-based scheduler

| *T1* | *T2* | *T3* | *A* | *B* | *C* |
|---|---|---|---|---|---|
| 200 | 150 | 175 | RT=0 | RT=0 | RT=0 |
| | | | WT=0 | WT=0 | WT=0 |
| r1(B); | | | | RT=200 | |
| | r2(A); | | RT=150 | | |
| | | r3(C); | | | RT=175 |
| w1(B); | | | | WT=200 | |
| w1(A); | | | WT=200 | | |
| | w2(C); | | | | |
| | **Abort;** | | | | |
| | | w3(A); | | | |

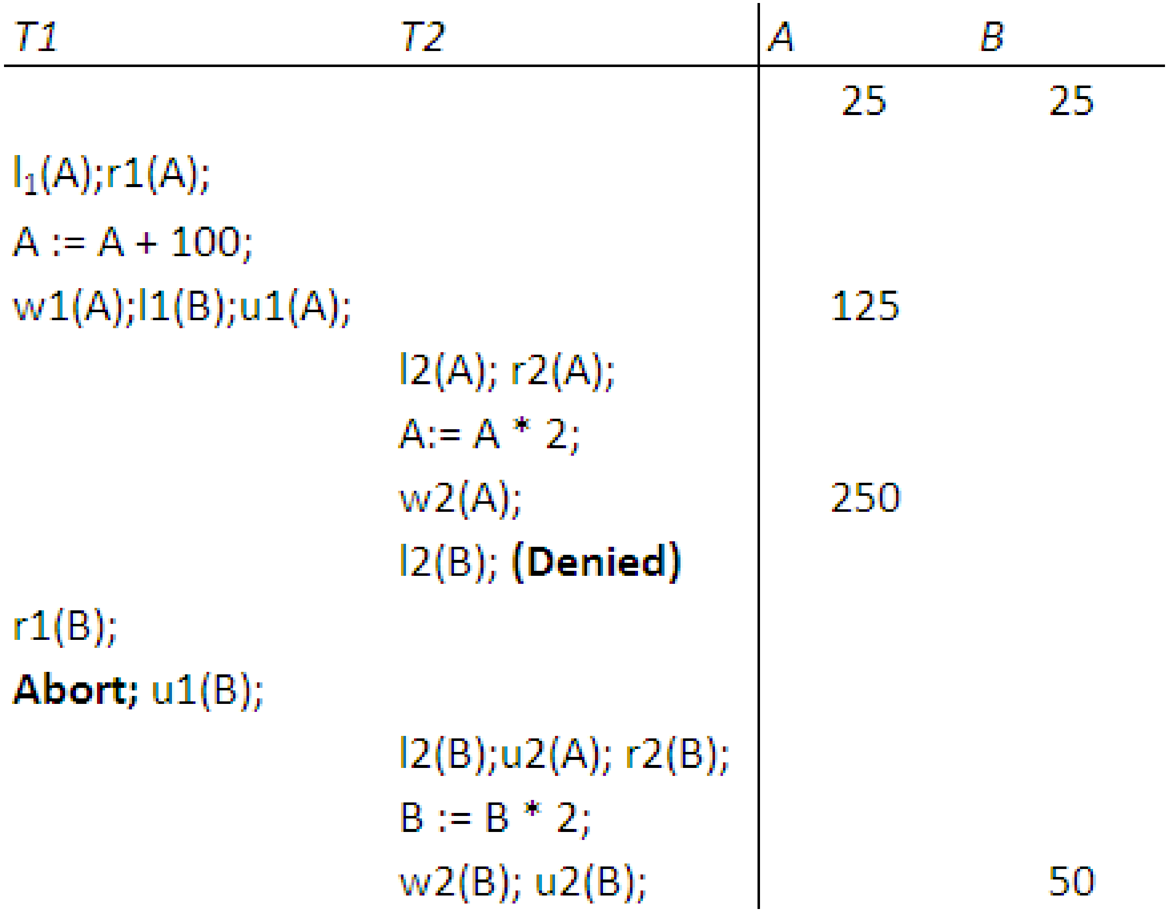
Neither aborted nor written


Read too late


Write too late

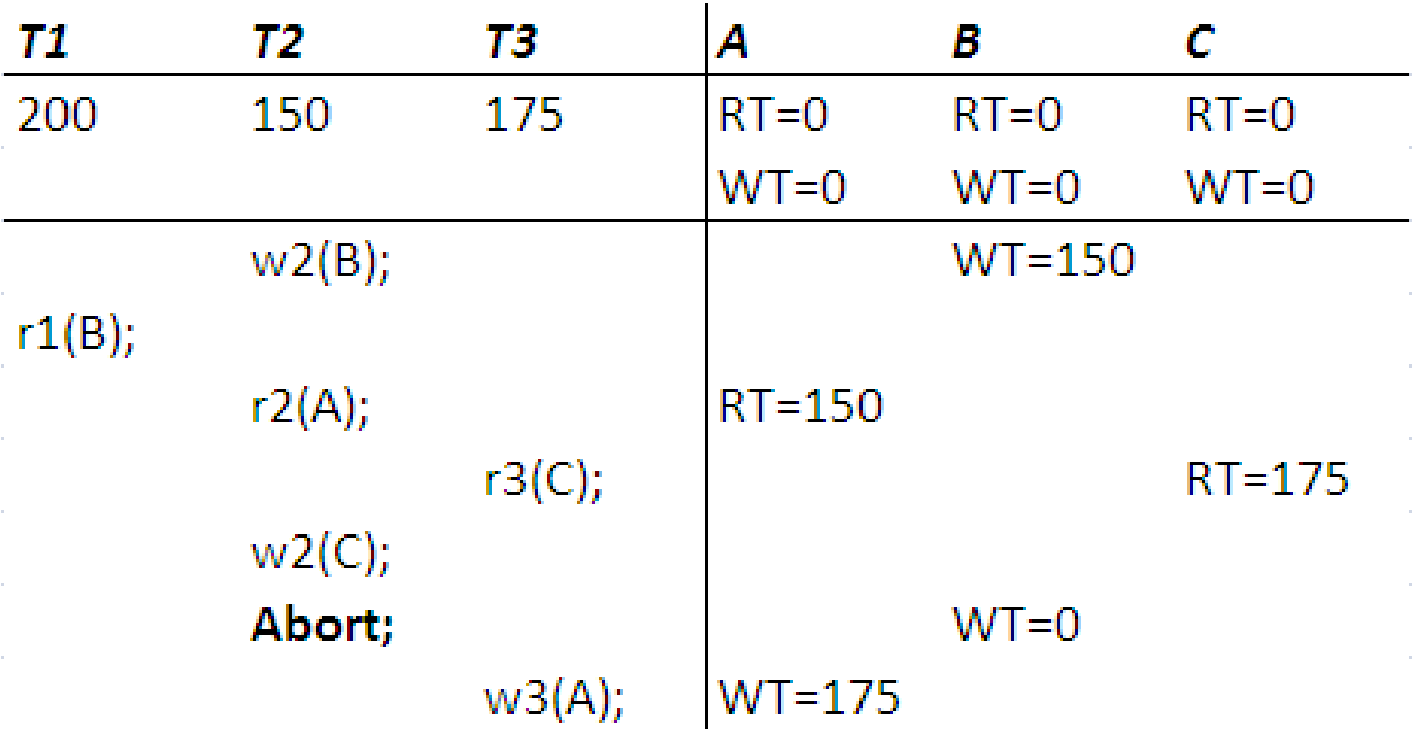
# Concurrency Control & Recovery

# Dirty Data Problem



| *T1* | *T2* | *A* | *B* |
|---|---|---|---|
| | | 25 | 25 |
| $l_1$(A);r1(A); | | | |
| A := A + 100; | | | |
| w1(A);l1(B);u1(A); | | 125 | |
| | l2(A); r2(A); | | |
| | A:= A * 2; | | |
| | w2(A); | 250 | |
| | l2(B); **(Denied)** | | |
| r1(B); | | | |
| **Abort;** u1(B); | | | |
| | l2(B);u2(A); r2(B); | | |
| | B := B * 2; | | |
| | w2(B); u2(B); | | 50 |

# Dirty Data Problem



| *T1* | *T2* | *T3* | *A* | *B* | *C* |
|---|---|---|---|---|---|
| 200 | 150 | 175 | RT=0 | RT=0 | RT=0 |
|  |  |  | WT=0 | WT=0 | WT=0 |
|  | w2(B); |  |  | WT=150 |  |
| r1(B); |  |  |  |  |  |
|  | r2(A); |  | RT=150 |  |  |
|  |  | r3(C); |  |  | RT=175 |
|  | w2(C); |  |  |  |  |
|  | **Abort;** |  |  | WT=0 |  |
|  |  | w3(A); | WT=175 |  |  |

Using timestamp scheduler with no commit bit

# How to Solve Dirty Data Problem?

- Cascading rollback
- Recoverable Schedules

# How to Solve Dirty Data Problem?

- Cascading rollback (Bad ☹)
- Recoverable Schedules

# Recoverable Schedules

- A schedule is recoverable if each transaction commits only after each transaction from which it has read has committed

$S_1$: $w_1(A)$; $w_1(B)$; $w_2(A)$;$r_2(B)$; $c_1$;$c_2$;

S2: $w_2(A)$; $w_1(B)$; $w_1(A)$;$r_2(B)$; $c_1$;$c_2$;

S3: $w_1(A)$; $w_1(B)$; $w_2(A)$;$r_2(B)$; $c_2$;$c_1$;

# Recoverable Schedules

- A schedule is recoverable if each transaction commits only after each transaction from which it has read has committed

$S_1$: $w_1(A)$; $w_1(B)$; $w_2(A)$;$r_2(B)$; $c_1$;$c_2$;

Recoverable
Serializable

S2: $w_2(A)$; $w_1(B)$; $w_1(A)$;$r_2(B)$; $c_1$;$c_2$;

Recoverable
Not serializable

S3: $w_1(A)$; $w_1(B)$; $w_2(A)$;$r_2(B)$; $c_2$;$c_1$;

Serializable
Not recoverable

# Avoids Cascading Rollback (ACR) Schedule

- A schedule S avoids cascading rollback if each transaction may *read* only those values written by committed transactions.

# Strict Schedule

- A schedule S is strict if each transaction may *read and write* only items previously written by committed transactions.

# Examples

S is recoverable if each transaction *commits* only after all transactions from which it read have committed.

- Recoverable
  - – $w_1(A)$ $w_1(B)$ $w_2(A)$ $r_2(B)$ $c_1$ $c_2$

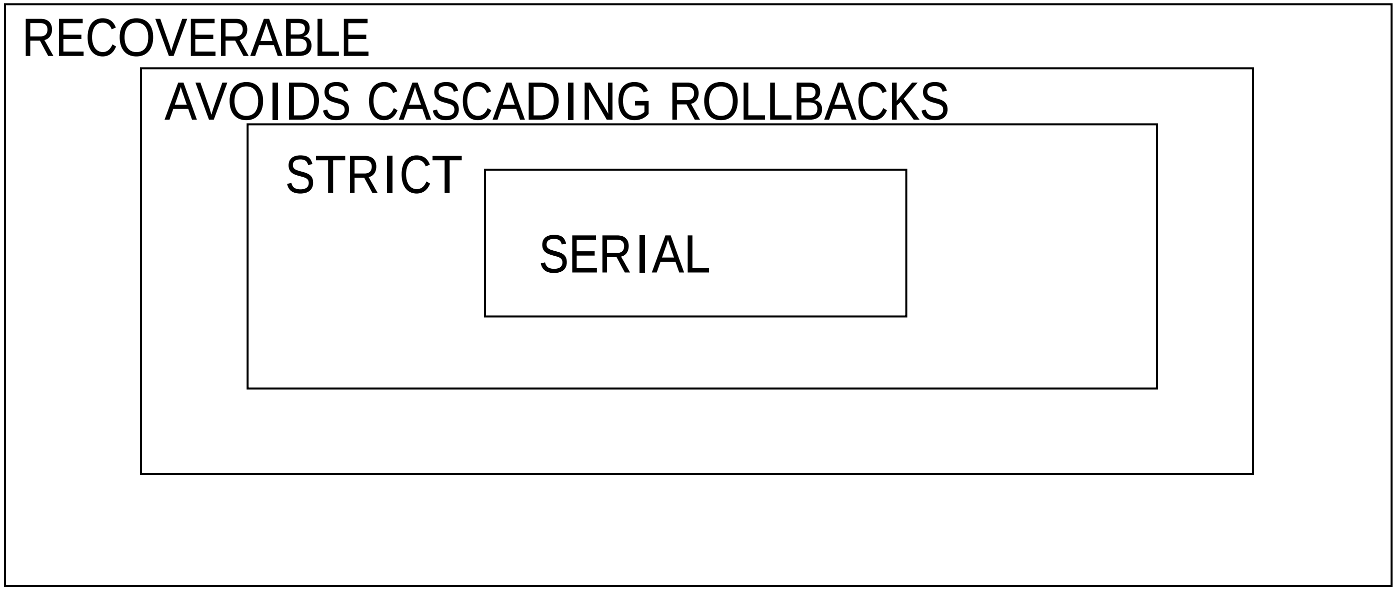
S avoids cascading rollback if each transaction may *read* only those values written by committed transactions.

- Avoids Cascading Rollback
  - – $w_1(A)$ $w_1(B)$ $w_2(A)$ $c_1$ $r_2(B)$ $c_2$

A schedule S is strict if each transaction may read and write only items previously written by committed transactions.

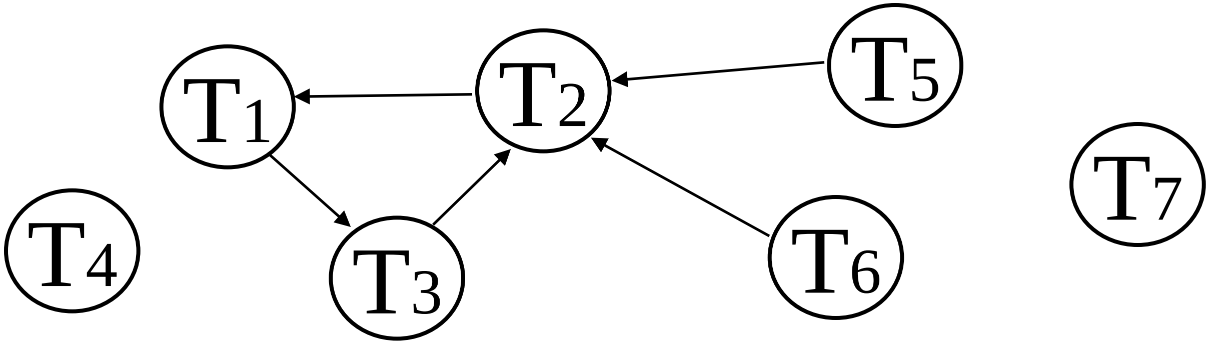
- Strict
  - – $w_1(A)$ $w_1(B)$ $c_1$ $w_2(A)$ $r_2(B)$ $c_2$

# Schedules



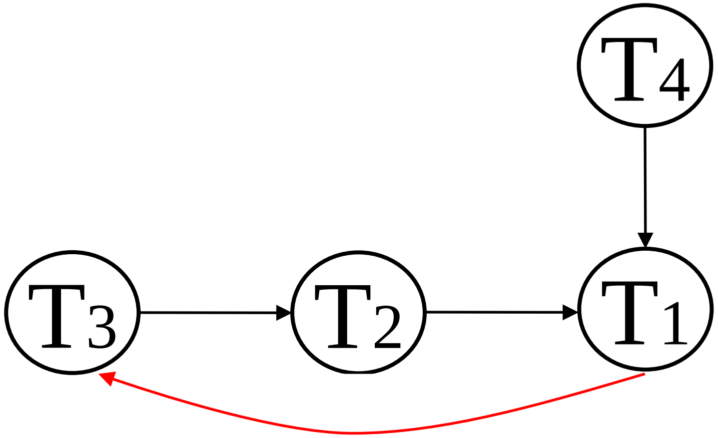
# Deadlock Detection

- Build Wait-For graph
- Use lock table structures
- Build incrementally or periodically
- When cycle found, rollback victim



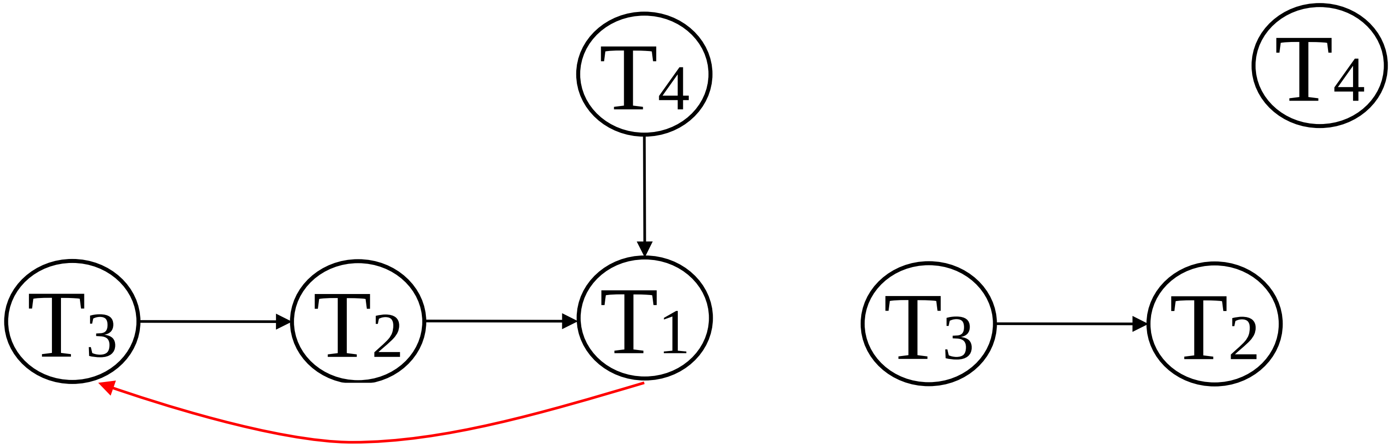
# Deadlock Detection: Wait-for Graph

| ***T1*** | ***T2*** | ***T3*** | ***T4*** |
|---|---|---|---|
| l1(A); r1(A); | | | |
| | l2(C);r2(C); | | |
| | | l3(B);r3(B); | |
| | | | l4(D);r4(D); |
| | l2(A); **Denied** | | |
| | | l3(C);**Denied** | |
| | | | l4(A); **Denied** |
| l1(B); **Denied** | | | |



# Deadlock Detection: Wait-for Graph

| *T1* | *T2* | *T3* | *T4* |
|---|---|---|---|
| l1(A); r1(A); | | | |
| | l2(C);r2(C); | | |
| | | l3(B);r3(B); | |
| | | | l4(D);r4(D); |
| | l2(A); **Denied** | | |
| | | l3(C);**Denied** | |
| | | | l4(A); **Denied** |
| l1(B); **Denied** | | | |



# Deadlock Detection: Timeout

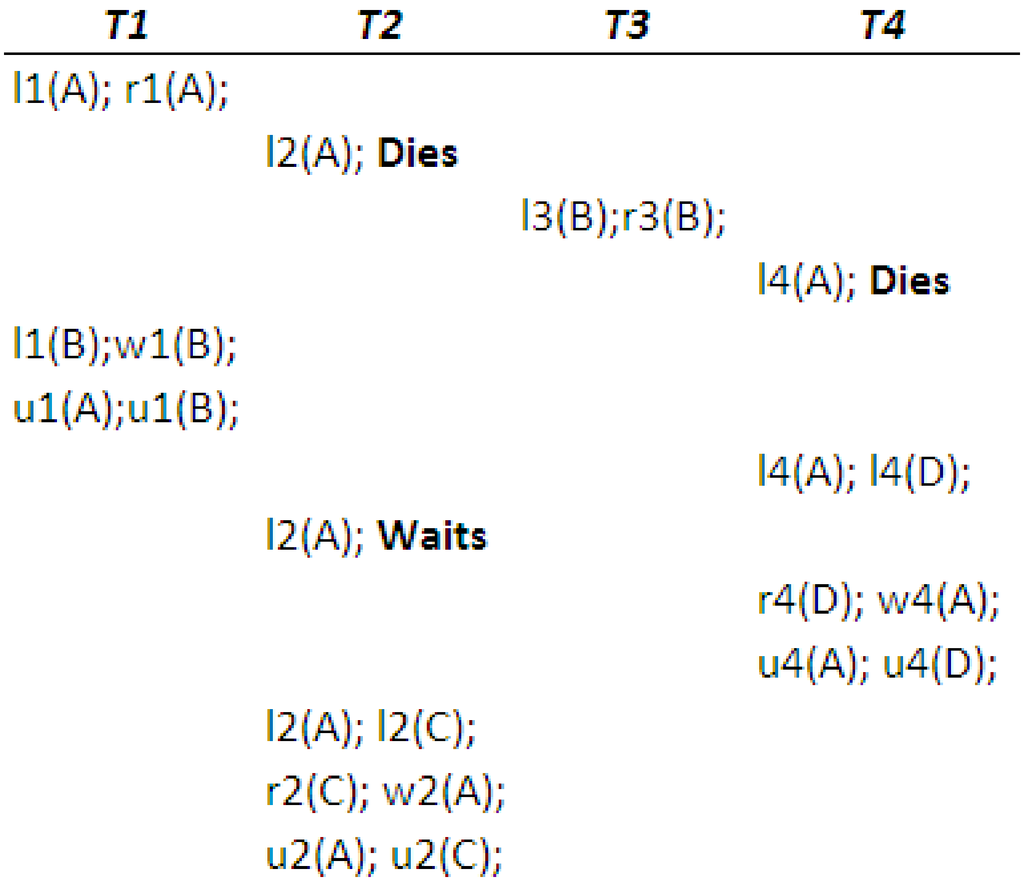
- If transaction waits more than L sec., roll it back!
- Simple scheme
- Hard to select L

# Deadlock Detection: Wait-die

- Transactions given a timestamp when they arrive …. ts($T_i$)
- $T_i$ can only wait for $T_j$ if ts($T_i$)< ts($T_j$)

  …else die

# Wait-die



| T1 | T2 | T3 | T4 |
|---|---|---|---|
| l1(A); r1(A); | | | |
| | l2(A); **Dies** | | |
| | | l3(B);r3(B); | |
| | | | l4(A); **Dies** |
| l1(B);w1(B); | | | |
| u1(A);u1(B); | | | |
| | | | l4(A); l4(D); |
| | l2(A); **Waits** | | |
| | | | r4(D); w4(A); |
| | | | u4(A); u4(D); |
| | l2(A); l2(C); | | |
| | r2(C); w2(A); | | |
| | u2(A); u2(C); | | |

# Starvation with Wait-Die

- When transaction dies, re-try later with what timestamp?
  - original timestamp
  - new timestamp (time of re-submit)

# Starvation with Wait-Die

- Resubmit with original timestamp
- Guarantees no starvation
  - Transaction with oldest ts never dies
  - A transaction that dies will eventually have oldest ts and will complete…

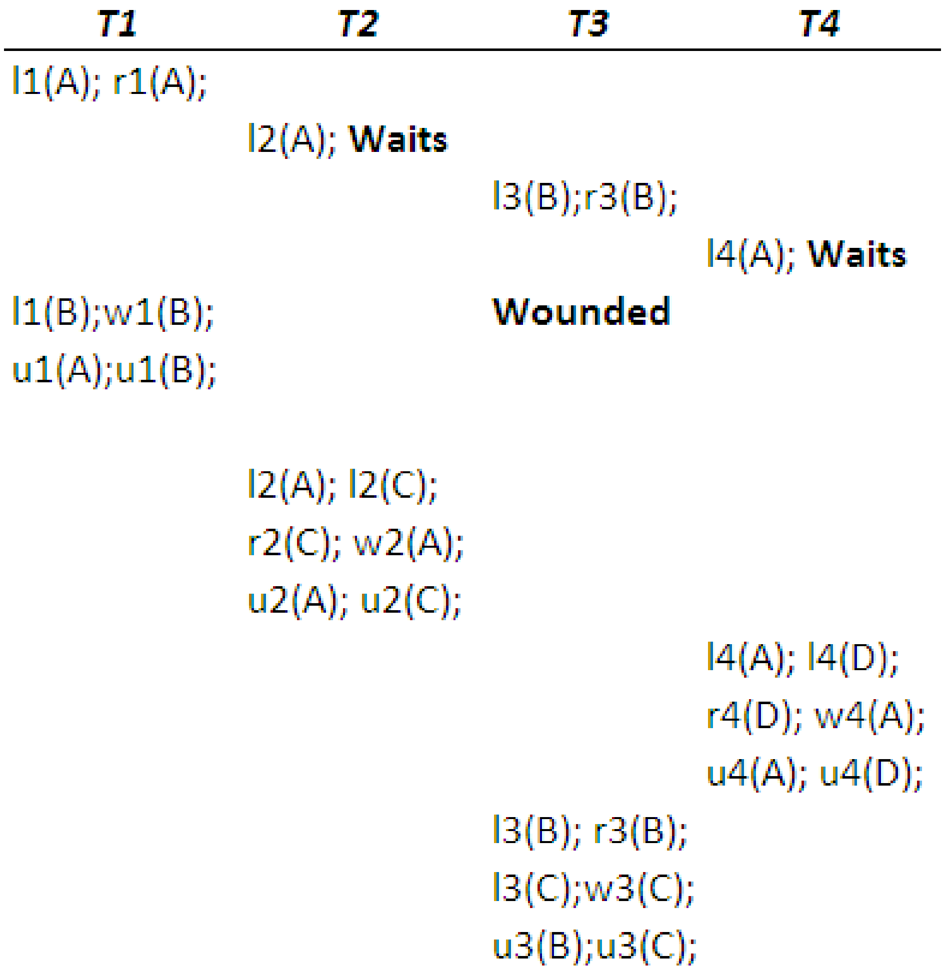
# Deadlock Detection: Wound-wait

- Transactions given a timestamp when they arrive ... ts($T_i$)
- $T_i$ wounds $T_j$ if ts($T_i$) < ts($T_j$)

  else $T_i$ waits

“Wound”: $T_j$ rolls back and gives lock to $T_i$

# Wound-wait



| ***T1*** | ***T2*** | ***T3*** | ***T4*** |
|---|---|---|---|
| l1(A); r1(A); | | | |
| | l2(A); **Waits** | | |
| | | l3(B);r3(B); | |
| | | | l4(A); **Waits** |
| l1(B);w1(B); | | **Wounded** | |
| u1(A);u1(B); | | | |
| | l2(A); l2(C); | | |
| | r2(C); w2(A); | | |
| | u2(A); u2(C); | | |
| | | | l4(A); l4(D); |
| | | | r4(D); w4(A); |
| | | | u4(A); u4(D); |
| | | l3(B); r3(B); | |
| | | l3(C);w3(C); | |
| | | u3(B);u3(C); | |

# Starvation with Wound-Wait

- When transaction dies, re-try later with what timestamp?
  - original timestamp
  - new timestamp (time of re-submit)

The END.