# CS 347:
# Distributed Databases and Transaction Processing
# **Notes06: Concurrency Control**

Hector Garcia-Molina

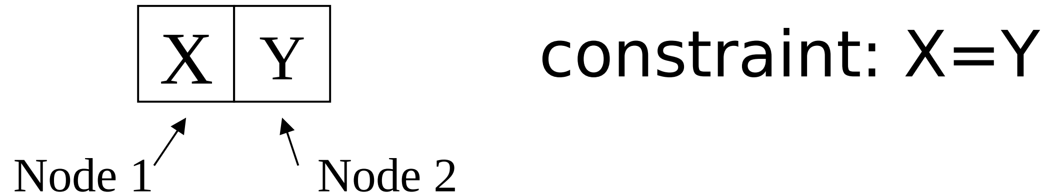
# Overview

- Concurrency Control
  - Schedules and Serializability
  - Locking
  - Timestamp Control
- Failure Recovery
  - next set of notes...

# Schedule

- Just like in a centralized system, a schedule represents how a set of transactions were executed
- Schedules may be “good” or “bad” (preserve constraints)

# Example




constraint: X=Y

| | $T_1$ | | $T_2$ |
|---|---|---|---|
| 1 | $(T_1)$ a ← X | 5 | $(T_2)$ c ← X |
| 2 | $(T_1)$ X ← a+100 | 6 | $(T_2)$ X ← 2c |
| 3 | $(T_1)$ b ← Y | 7 | $(T_2)$ d ← Y |
| 4 | $(T_1)$ Y ← b+100 | 8 | $(T_2)$ Y ← 2d |

↓ Precedence relation

# Schedule S1

| (node X) | | | (node Y) | | |
|---|---|---|---|---|---|
| 1 | ($T_1$) | a ← X | | | |
| 2 | ($T_1$) | X ← a+100 | | | |
| 5 | ($T_2$) | c ← X | 3 | ($T_1$) | b ← Y |
| 6 | ($T_2$) | X ← 2c | 4 | ($T_1$) | Y ← b+100 |
| | | | 7 | ($T_2$) | d ← Y |
| | | | 8 | ($T_2$) | Y ← 2d |

If X=Y=0 initially, X=Y=200 at end (always good?)

# Definition of Schedule

Let T= $\{T_1, T_2, T_n\}$ be a set of transactions.

A schedule S over T is a partial order with ordering relation $<_S$ where:

(1) $S = \bigcup_{i=1}^{N} T_i$

(2) $<_S \supseteq \bigcup_{i=1}^{N} <_i$

(3) for any two conflicting operations $p, q \in S$,
either $p <_S q$ or $q <_S p$

# Example

$(T_1)$ $r_1[X] \rightarrow W_1[X]$

$(T_2)$ $r_2[X] \rightarrow W_2[Y] \rightarrow W_2[X]$

$(T_3)$ $r_3[X] \rightarrow W_3[X] \rightarrow W_3[Y] \rightarrow W_3[Z]$

$r_2[X] \rightarrow W_2[Y] \rightarrow W_2[X]$

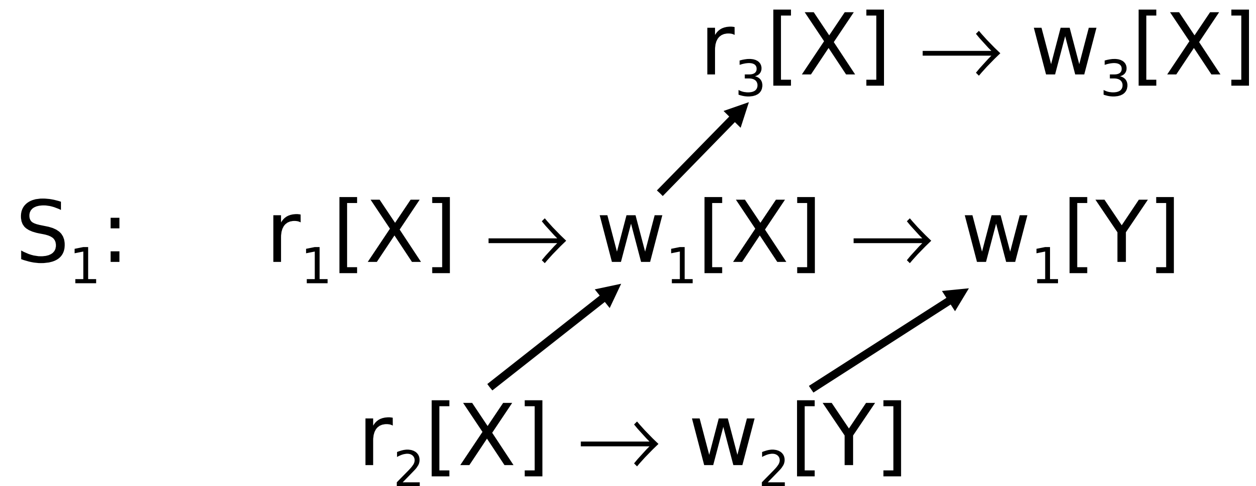
$S_1$: $r_3[Y] \rightarrow W_3[X] \rightarrow W_3[Y] \rightarrow W_3[Z]$

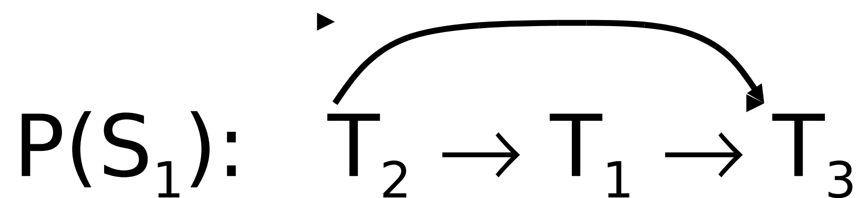
$r_1[X] \rightarrow W_1[X]$

# Definition of P(S)

- The precedence graph for schedule S, P(S), is a directed graph where
  - nodes: the transactions in S
  - edges: $T_i \rightarrow T_j$ is an edge IFF
    $\exists$ p $\in$ $T_i$, q $\in$ $T_j$ such that
    p, q conflict and p $<_S$ q

# Example



# Serializability Theorem

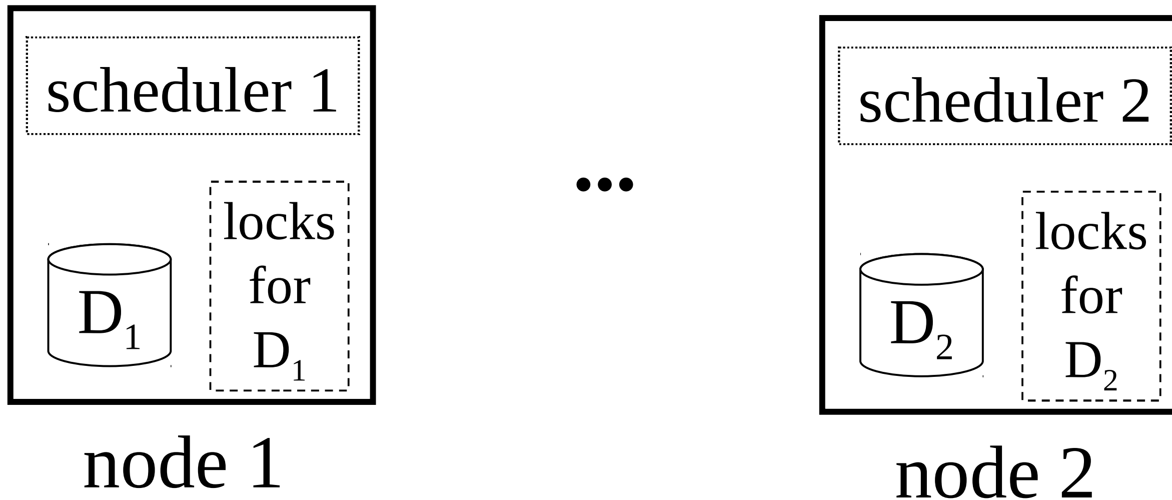
Theorem: A schedule S is serializable IFF P(S) is acyclic.

# Enforcing Serializability

- Locking
- Timestamps

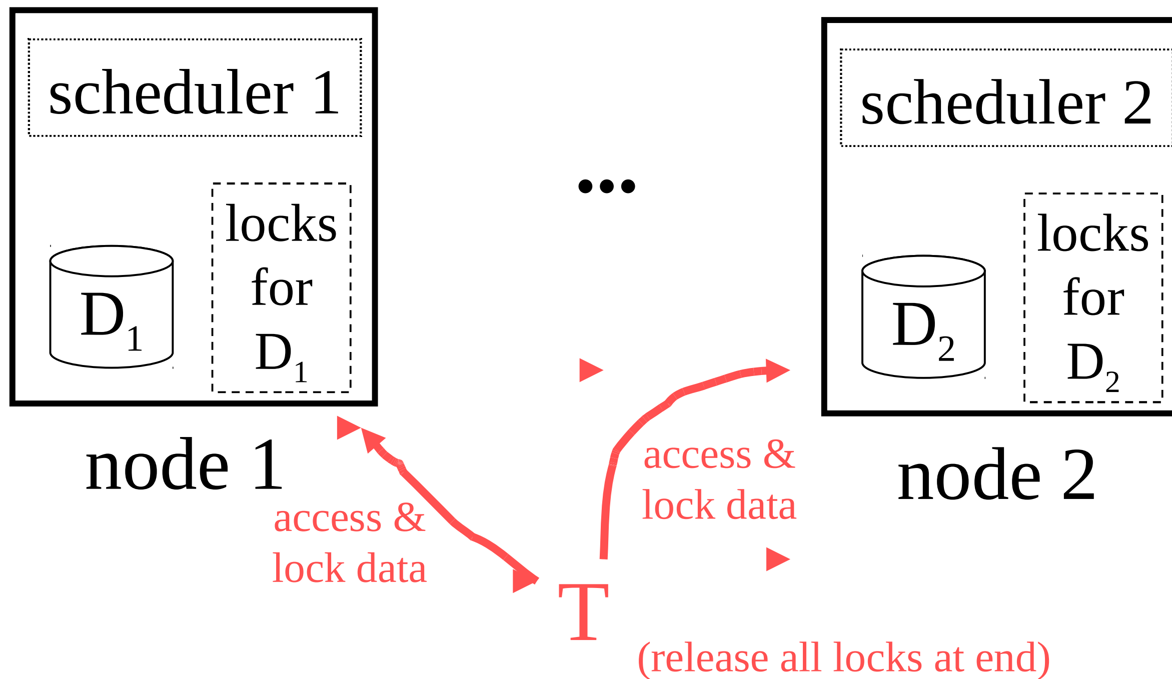
# Locking

- Just like in a centralized system...
- But with multiple lock managers



# Locking

- Just like in a centralized system...
- But with multiple lock managers



# Locking Rules

- Well-formed transactions
- Legal schedulers
- Two-phase transactions

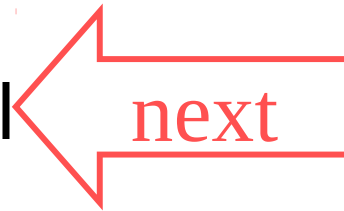
- These rules guarantee serializable schedules

# What about

- Locking in a shared-memory architecture?
- Locking in a shared-disks architecture?

# Overview

- Concurrency Control
  - Schedules and Serializability
  - Locking
  - Timestamp Control




- Failure Recovery
  - next set of notes...

# Timestamp Ordering Schedulers

- Basic idea:

  - assign timestamp as transaction begins

  - if $ts(T_1) < ts(T_2) \ldots < ts(T_n)$, then scheduler produces history equivalent to $T_1, T_2, T_3, T_4, \ldots T_n$

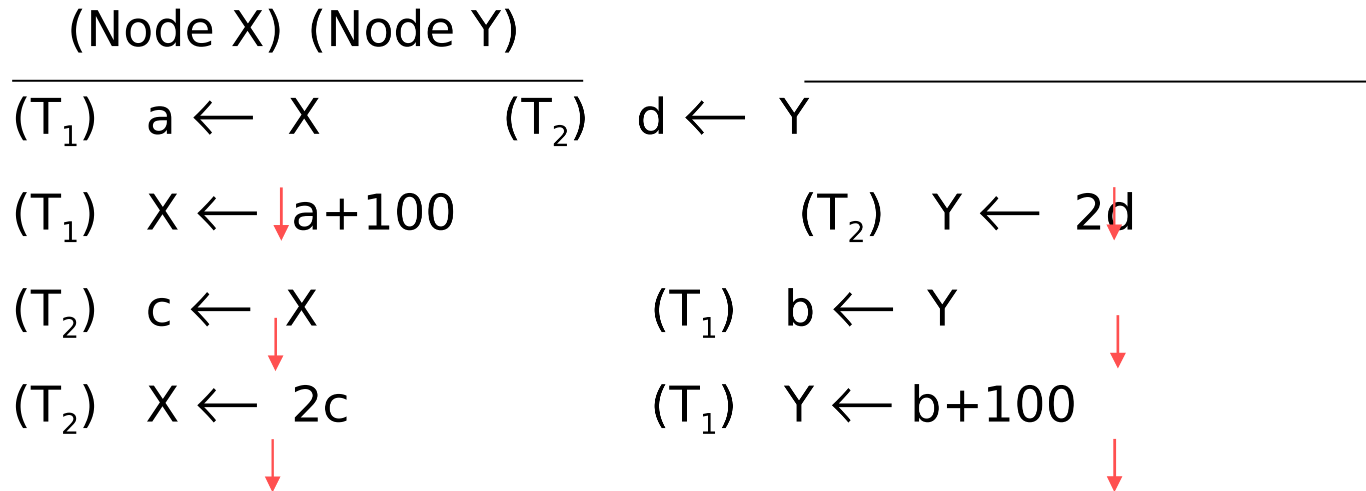
# TO Rule

If $p_i[x]$ and $q_j[x]$ are conflicting operations, then $p_i[x]$ is executed before $q_j[x]$

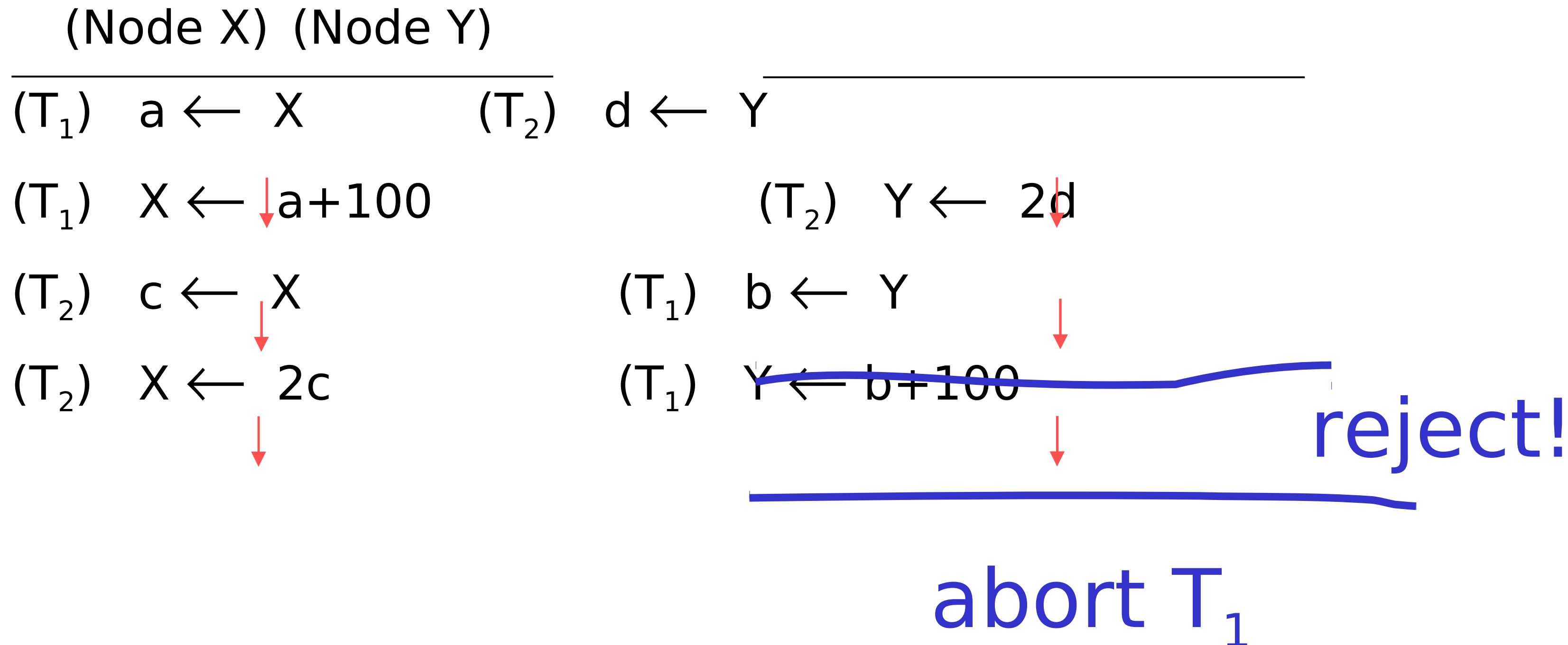
$(p_i[x] <_S q_j[x])$

IFF $ts(T_i) < ts(T_j)$

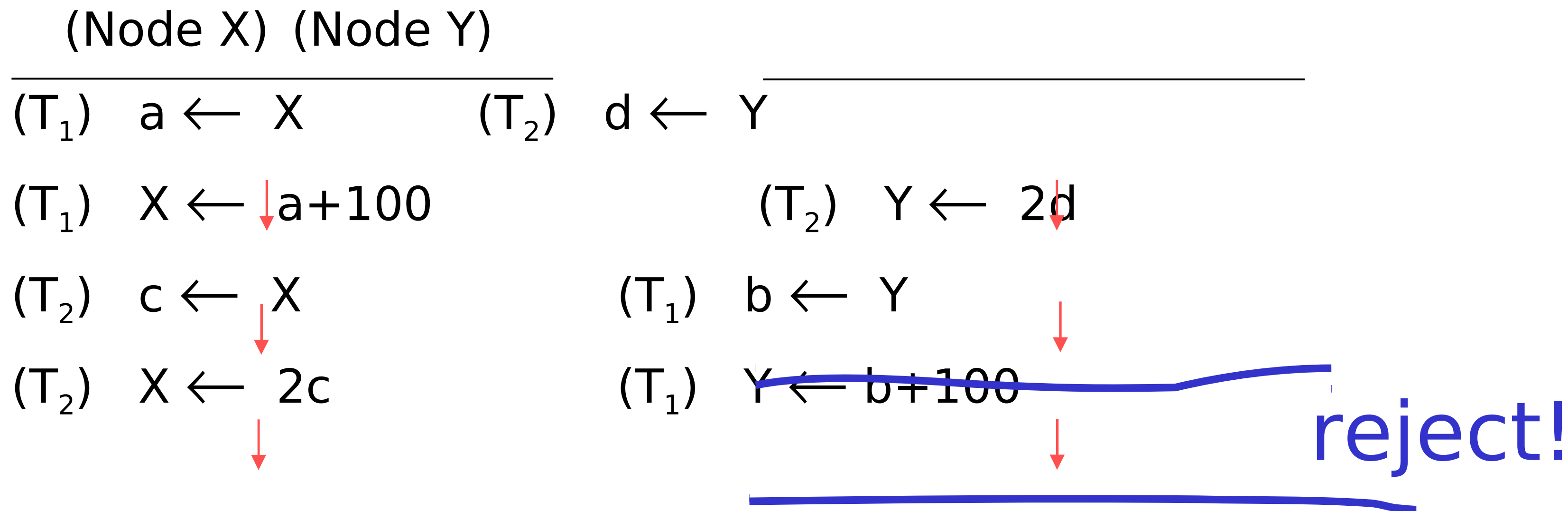
# Example: a non-serializable schedule $S_2$

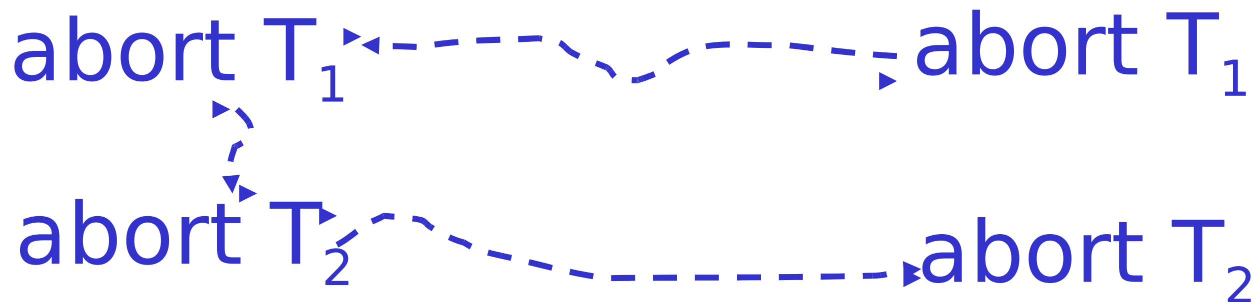


# Example: a non-serializable schedule $S_2$



# Example: a non-serializable schedule $S_2$



# Strict T.O.

- Lock written items until it is certain that writing transaction has been successful (avoid cascading rollbacks)

# Example Revisited

$ts(T_1) < ts(T_2)$

| (Node X) | | (Node Y) | |
|---|---|---|---|
| $(T_1)$ | $a \leftarrow X$ | $(T_2)$ | $d \leftarrow Y$ |
| $(T_1)$ | $X \leftarrow a+100$ | $(T_2)$ | $Y \leftarrow 2d$ |
| $(T_2)$ | $c \leftarrow X$ delay | $(T_1)$ | ~~$b \leftarrow Y$~~ reject! abort $T_1$ |

# Example Revisited

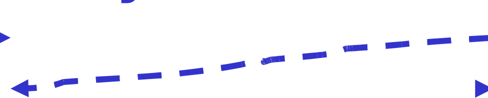
$ts(T_1) < ts(T_2)$

(Node X)

($T_1$) a ← X

($T_1$) X ← a+100

($T_2$) c ← X



delay

abort $T_1$

($T_2$) c ← X

($T_2$) X ← 2c

(Node Y)

($T_2$) d ← Y

($T_2$) Y ← 2d

($T_1$) b ← Y

reject!

abort $T_1$

# Enforcing T.O.

- For each data item X:

MAX_R[X]: maximum timestamp of a transaction that read X

MAX_W[X]: maximum timestamp of a transaction that wrote X

rL[X]: # of transactions currently reading X (0,1,2,…)

wL[X]: # of transactions currently writing X (0 or 1)

# T.O. Scheduler - Part 1

$r_i$[X] arrives

IF $ts(T_i)$ < MAX_W[X] THEN ABORT $T_i$

ELSE [ IF $ts(T_i)$ > MAX_R[X] THEN MAX_R[X] $\leftarrow$ $ts(T_i)$;

IF queue is empty AND wL[X] = 0 THEN

[rL[X] $\leftarrow$ rL[X]+1;  START READ OF X]

ELSE add (r, $T_i$) to queue]

# T.O. Scheduler - Part 2

$W_i[X]$ arrives

IF $ts(T_i) < MAX_W[X]$ OR $ts(T_i) < MAX_R[X]$
THEN ABORT $T_i$

ELSE [ $MAX_W[X] \leftarrow ts(T_i)$;

IF queue is empty AND $wL[X]=0$ AND $rL[X]=0$

THEN [$wL[X] \leftarrow 1$; WRITE X;
WAIT FOR $T_i$ TO FINISH ]

ELSE add (w, $T_i$) to queue ]

# T.O. Scheduler - Part 3

When o finishes (o is r or w) on X

oL[X] $\leftarrow$ oL[X] - 1; NDONE $\leftarrow$ TRUE

WHILE NDONE DO

[ let head of queue be (q, $T_j$); *(smallest timestamp)*

IF q=w AND rL[X]=0 AND wL[X]=0 THEN

[remove (q,$T_j$); wL[X] $\leftarrow$ 1;

WRITE X AND WAIT FOR $T_j$ TO FINISH ]

ELSE IF q=r AND wL[X]=0 THEN

[remove (q,$T_j$); rL[X] $\leftarrow$ rL[X] +1; START READ OF X]

ELSE NDONE $\leftarrow$ FALSE ]

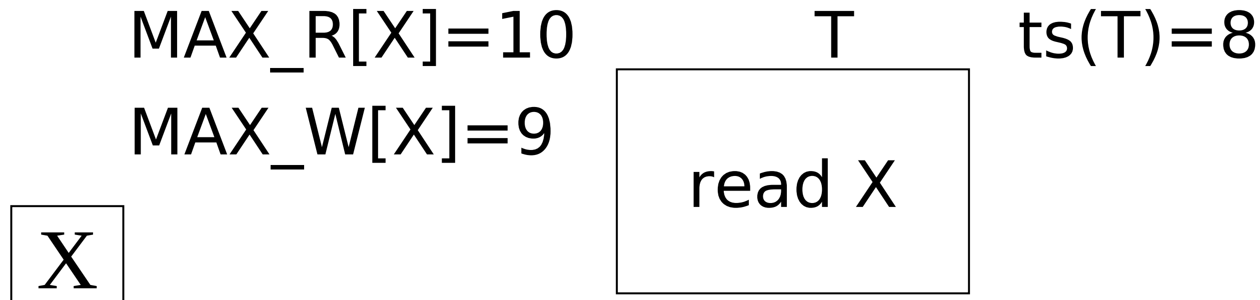
# Note about the code

for reads: [rL[X] $\leftarrow$ rL[X]+1; START READ OF X]

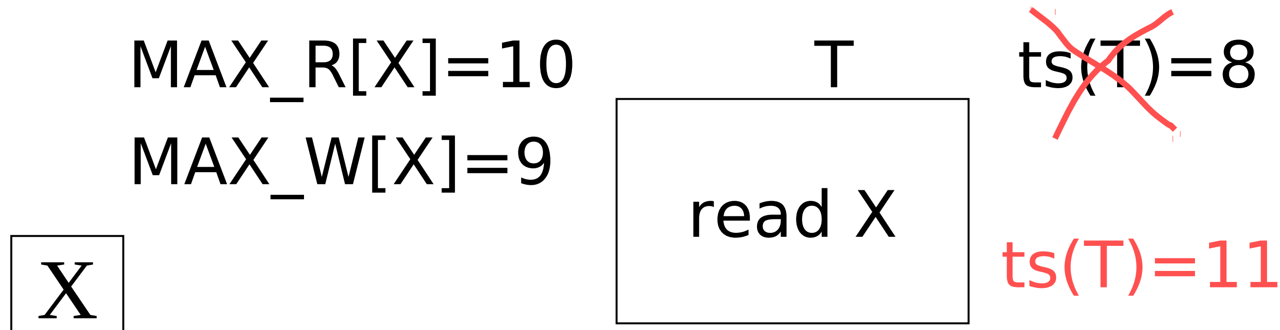
for writes: [wL[X] $\leftarrow$ 1; WRITE X;
WAIT FOR $T_i$ TO FINISH ]

Meaning: In Part 3, the end of a write is only processed when all writes for its transaction have completed.

- ☛ If a transaction is aborted, it must be retired with a new, larger timestamp

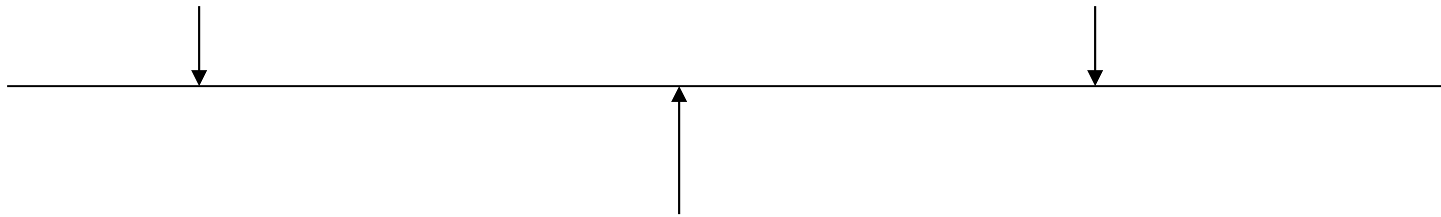


- If a transaction is aborted, it must be retired with a new, larger timestamp




- Starvation possible

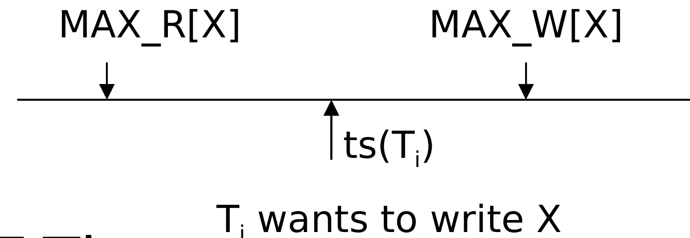
# Thomas Write Rule




$ts(T_i)$

$T_i$ wants to write X

# Change in T.O. Scheduler:




## When Wi[X] arrives

IF ts(Ti)<MAX_R[X] THEN ABORT Ti

ELSE IF ts(Ti)<MAX_W[X] THEN

[IGNORE THIS WRITE (tell Ti it was OK)]

ELSE [process write as before…

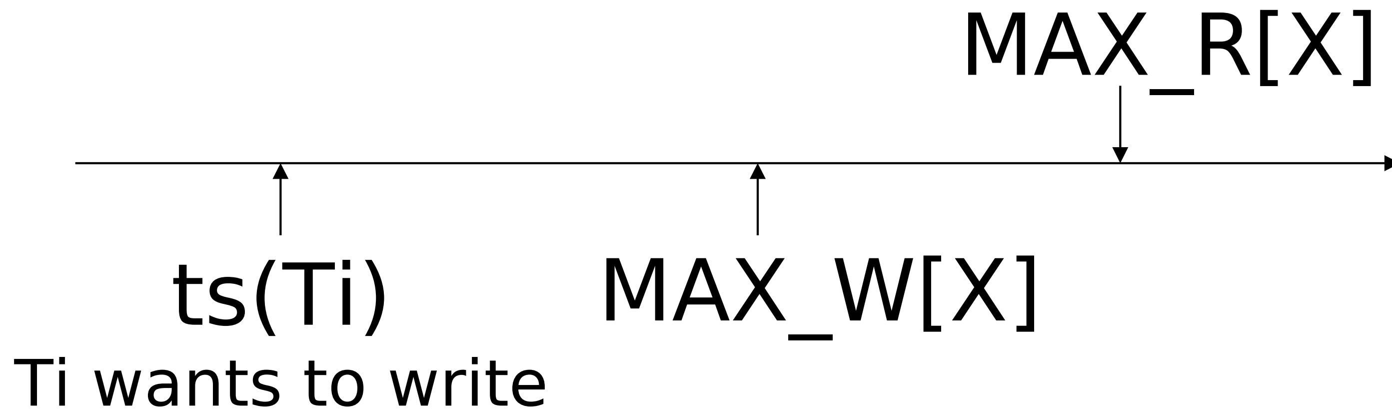
MAX_W[X] $\leftarrow$ ts(Ti);

IF queue is empty AND wL[X]=0 AND rL[X] =0 THEN

[wL[X] $\leftarrow$ 1; WRITE X and WAIT FOR Ti TO FINISH]

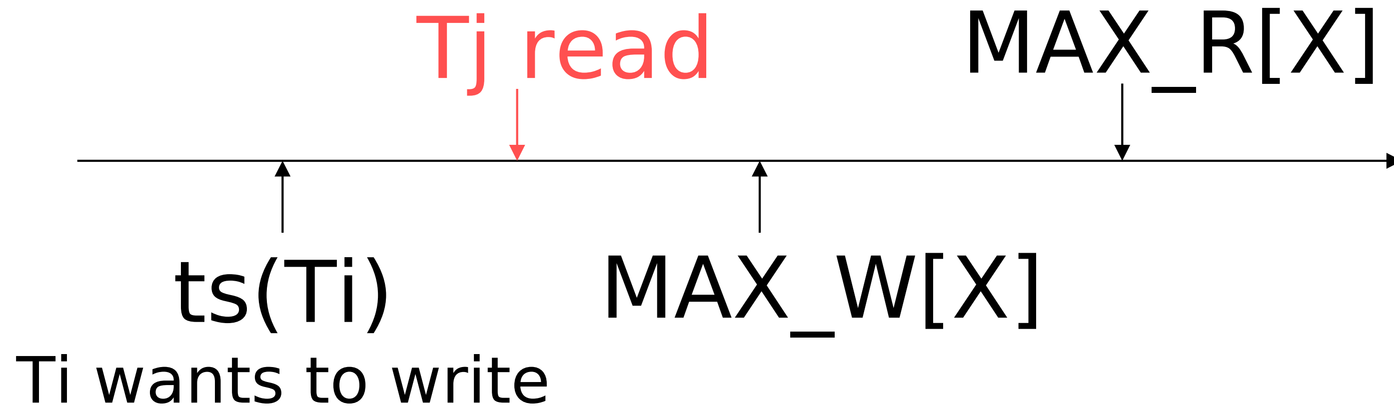
ELSE add (W, Ti) to queue]

# Question




- Why can't we let Ti go ahead?

# Question



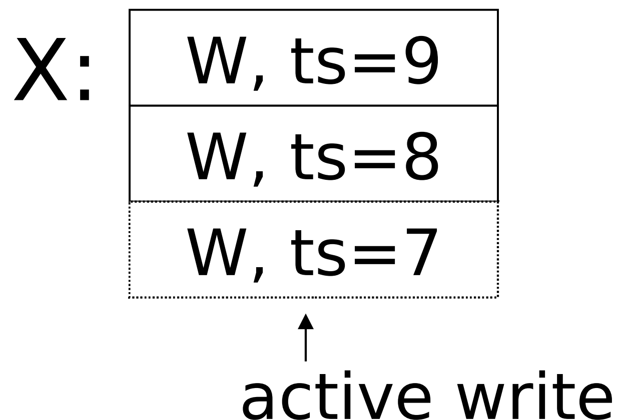

- Why can't we let Ti go ahead?

☞ MAX_R[X] is only the latest read; there could be a Tj read as shown...

# Optimizations

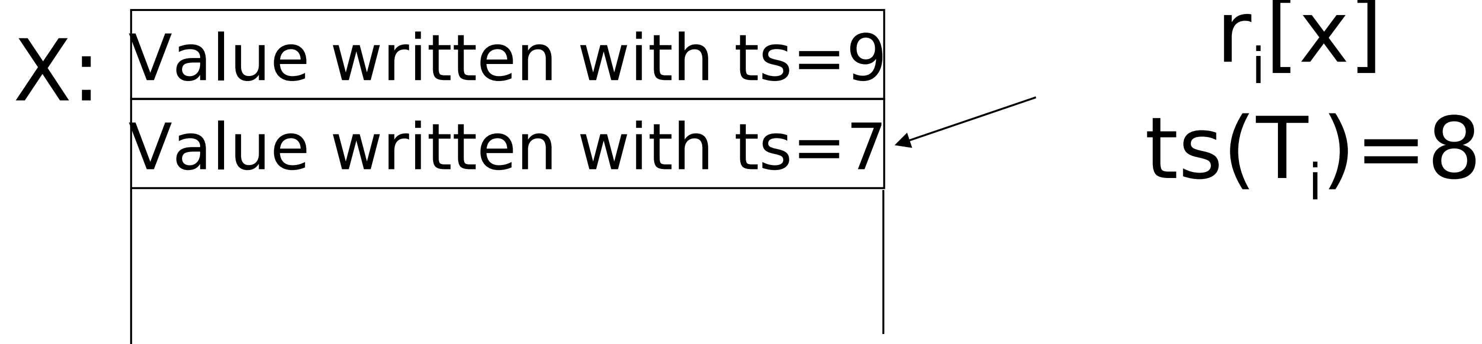
- Update MAX_R, MAX_W when action executed, not when action put on queue

Example: MAX_W[X]=9 or 7?



# Optimizations

- Use multiple versions of data



# 2PL ≠ TO

$T_1$: $w_1[Y]$

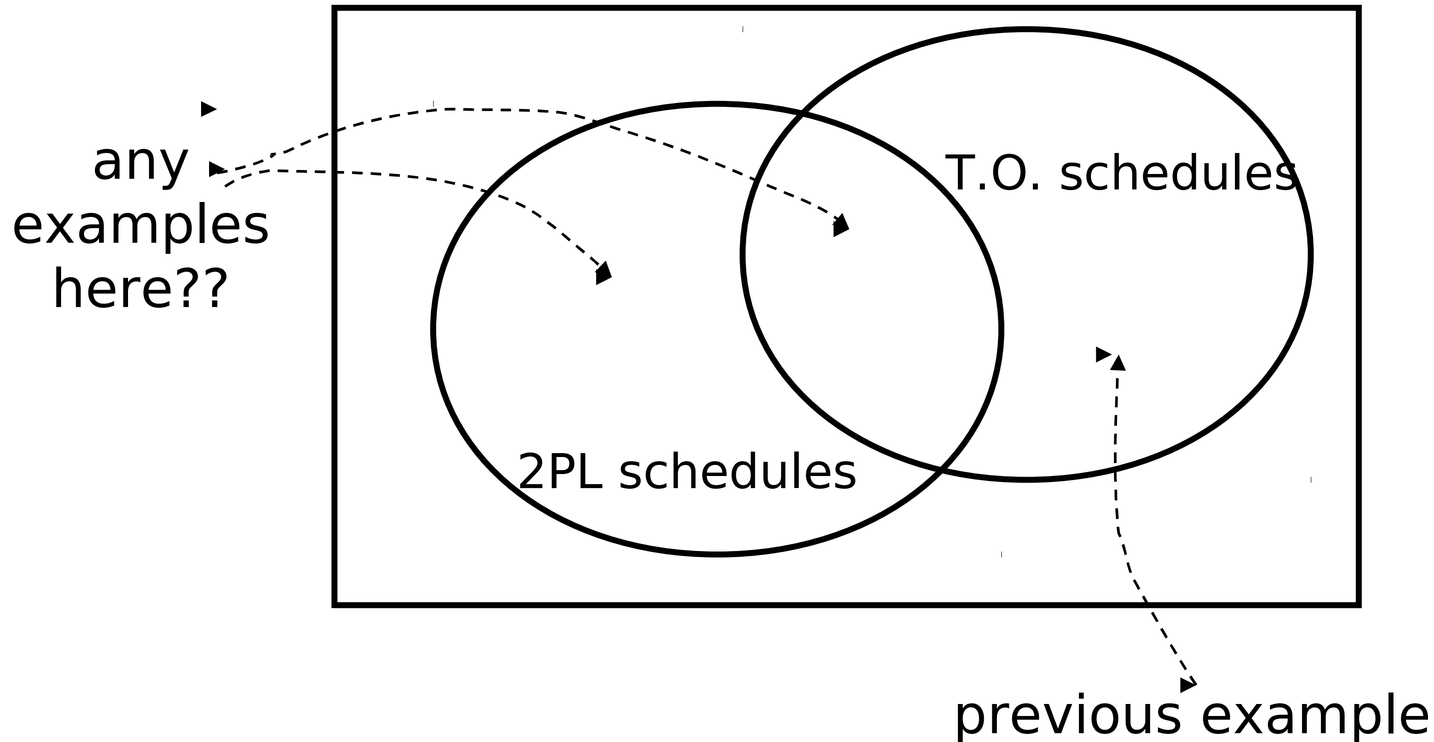
$T_2$: $r_2[X]$ $r_2[Y]$ $w_2[Z]$ $ts(T_1)<ts(T_2)<ts(T_3)$

$T_3$: $w_3[X]$

S: $r_2[X]$ $w_3[X]$ $w_1[Y]$ $r_2[Y]$ $w_2[Z]$

- S could be produced with T.O. but not with 2PL

# Are all 2PL schedules T.O.?



# Theorem

If S is a schedule representing an execution by a T.O. scheduler, then S is serializable

Proof:

1) say $T_i \rightarrow T_j$ in P(S)

$\Rightarrow \exists$ conflicting $p_i[x]$, $q_j[x]$ in S,

such that $p_i[x] <_S q_j[x]$

Then by T.O. rule, $ts(T_i) < ts(T_j)$

# Proof - Continued

2) Say there is a cycle
   $T_1 \rightarrow T_2 \rightarrow \ldots T_n \rightarrow T_1$ in P(S)
   then:
   $ts(T_1) < ts(T_2) < ts(T_3) \ldots$
   $< ts(T_n) < ts(T_1)$

   A contradiction!

3) So P(S) is acyclic

   $\Rightarrow$ S is serializable

# Timestamp management

| Item | data | MAX_R | MAX_W |
| --- | --- | --- | --- |
| X1 | | | |
| X2 | | | |
| . | | | |
| . | | | |
| . | | | |
| Xn | | | |

- too much space!
- more IO

# Timestamp cache

| Item | MAX_R | MAX_W |
| --- | --- | --- |
| X | | |
| Y | | |
| | | |
| Z | | |

tsMIN

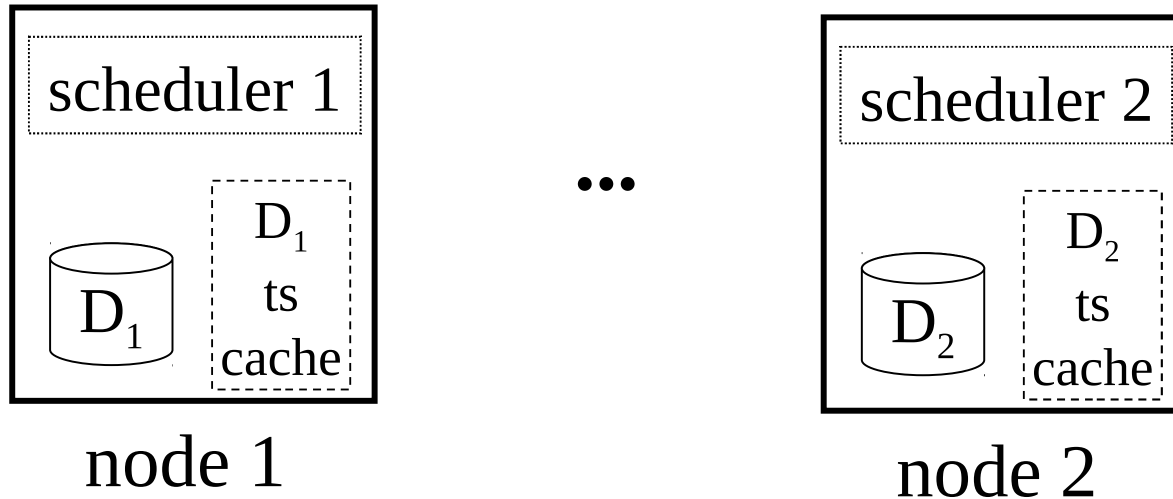
| |
| --- |
| |

- If transaction reads or writes X, make entry in cache for X (add row if not in)
- Periodically purge all items X with MAX_R[X] < tsMIN, MAX_W[X] < tsMIN and remember tsMIN (choose tsMIN ≈ current time - d)

# Timestamp cache

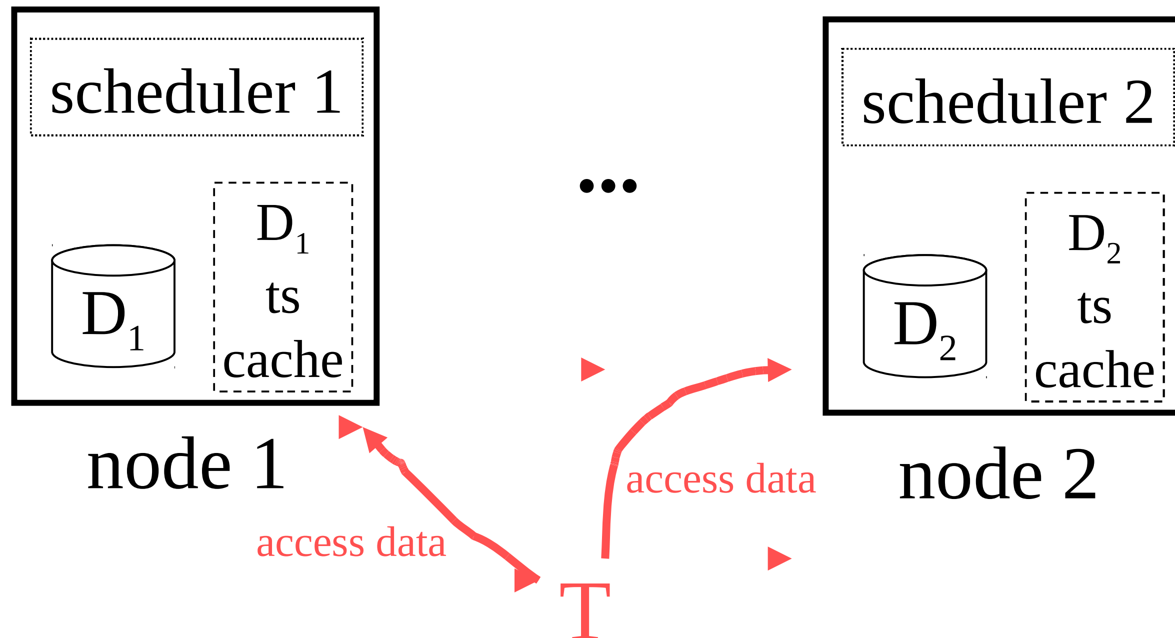
- To enforce T.O. rule for pi[X]
  - if X entry in cache,
    use MAX_R, MAX_W values in cache
  - else assume
    MAX_R[X]=tsMIN, MAX_W[X]=tsMIN
- Use hashing (on items) for cache
  (same as lock table)

# Distributed T.O. Scheduler




- Each scheduler is "independent"
- At end of transaction, signal all schedulers involved to release all wL[X] locks

# Distributed T.O. Scheduler




- Each scheduler is "independent"
- At end of transaction, signal all schedulers involved to release all wL[X] locks

# Summary

- 2PL - the most popular - useful in a distributed system- deadlocks possible - several variations
- T.O. - good for multiple versions

  - aborts more likely - no deadlocks - useful in a distributed sytem

- Others concurrency control schemes e.g., Certifiers, serialization graph testing

hard to implement in a distributed system

not very practical
need global data structure