More Transaction Processing

Background

- The concurrency control mechanisms studied to date – will work flawlessly, if there are no aborts.
- But, transactions need to be aborted for various reasons: failures/recovery, deadlock (as we'll see later), etc.
- Aborted transactions cause problems in the non-aborted transactions (next slide).

Topics

- Cascading rollback, Recoverable schedules, Strict schedules
- Deadlocks

Concurrency control & recovery

Example:

T_j	T_i
⋮	⋮
$W_j(A)$	⋮
	$R_i(A)$
⋮	Commit T_i
⋮	
Abort T_j	⋮

☛ Cascading rollback (Bad!)

- Schedule is conflict serializable
- But, not recoverable (because we need to abort a committed transaction T_i).

- Need to make "final" decision for each transaction:
  - commit decision - system guarantees transaction will or has completed, no matter what
  - abort decision - system guarantees transaction will or has been rolled back (has no effect)

To model this, two new actions:

- C_i - transaction T_i commits
- A_i - transaction T_i aborts

Reads and writes precede commit or abort.

- If $C_i \in T_i$, then $r_i(A) < C_i$
  $w_i(A) < C_i$
- If $A_i \in T_i$, then $r_i(A) < A_i$
  $w_i(A) < A_i$

- Also, one of Ci, Ai per transaction

Back to example:

T_j	T_i
⋮	⋮
$W_j(A)$	
	$R_i(A)$
⋮	
	⋮
	C_i ← can we commit here?

Definition

T_i **reads from** T_j in S ($T_j \Rightarrow_S T_i$) if

(1) $w_j(A) <_S r_i(A)$

(2) $a_j \not<_S r_i(A)$ ($\not<$: does not precede)

(3) If $w_j(A) <_S w_k(A) <_S r_i(A)$ then

$a_k <_S r_i(A)$

Recoverable Schedules

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

Formally,

Schedule S is recoverable if whenever $Tj \Rightarrow_S Ti$ and $j \neq i$ and $Ci \in S$ then $Cj <_S Ci$

The above ensures that committed transaction would never need to be aborted.

ACR Schedules

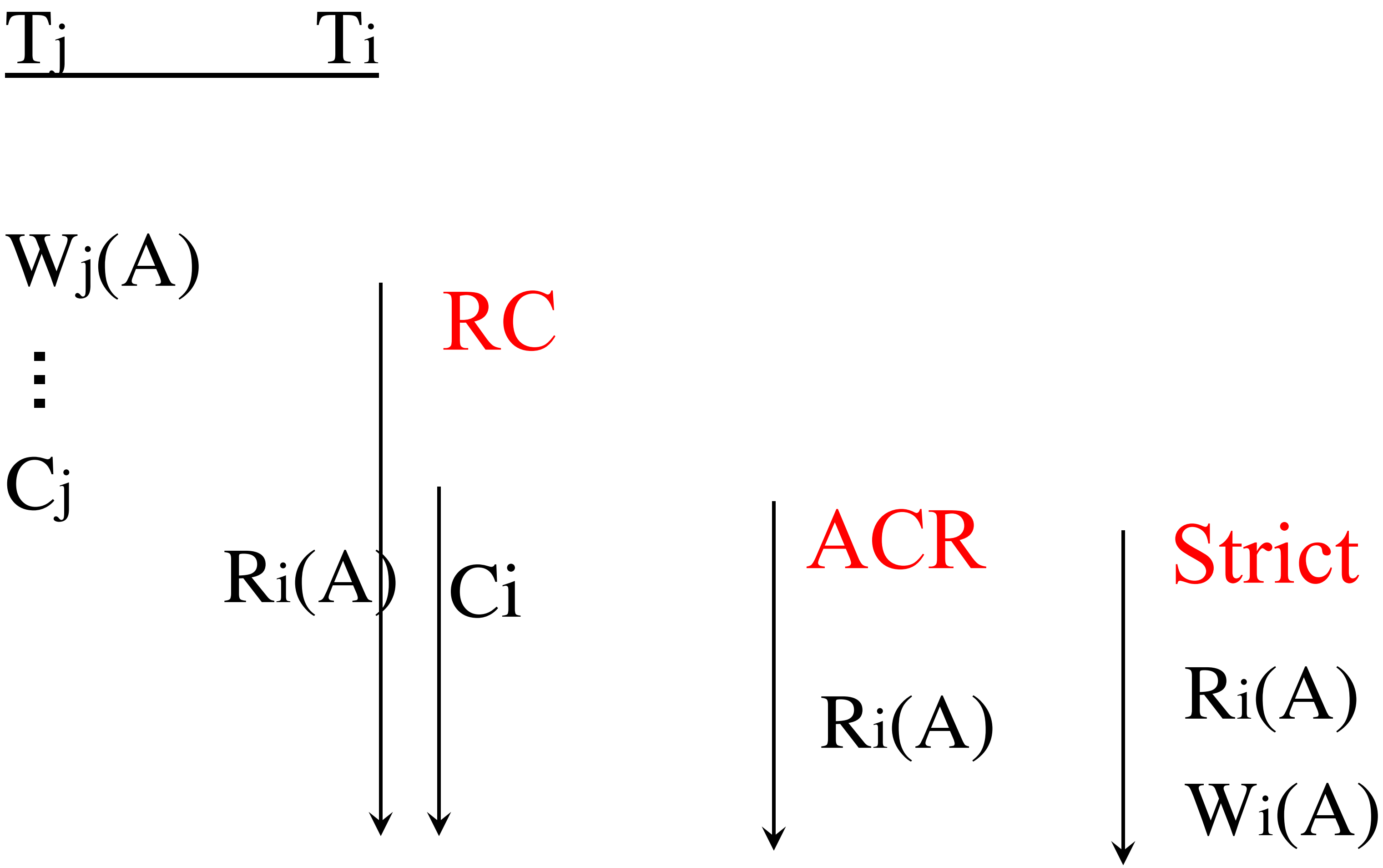
- Recoverable schedules may still result in cascading rollback of other uncommitted transactions.

- Schedule S is ACR, i.e., *avoids cascading rollback*, if each transaction in S may *read* only committed values (i.e., those written by committed transactions).

Strict 2PL Locking

- One way to achieve ACR (and recoverable) schedules through locking is:
  - Retain the "write" locks until the very end (i.e., unlock just before committing).
  - This is called Strict locking. [Strict-2PL locking is different]
  - Yields Strict schedules. Herein, essentially, each transaction reads-from and writes-into only committed values.

Recoverable, ACR, and Strict Schedules



Examples

- Recoverable:
  - ➢ $w_1(A)\ w_1(B)\ \ w_2(A)\ r_2(B)\ \ \ c_1\ c_2$
- Avoids Cascading Rollback:
  - ➢ $w_1(A)\ w_1(B)\ \ w_2(A)\ \ c_1\ \ r_2(B)\ \ \ c_2$

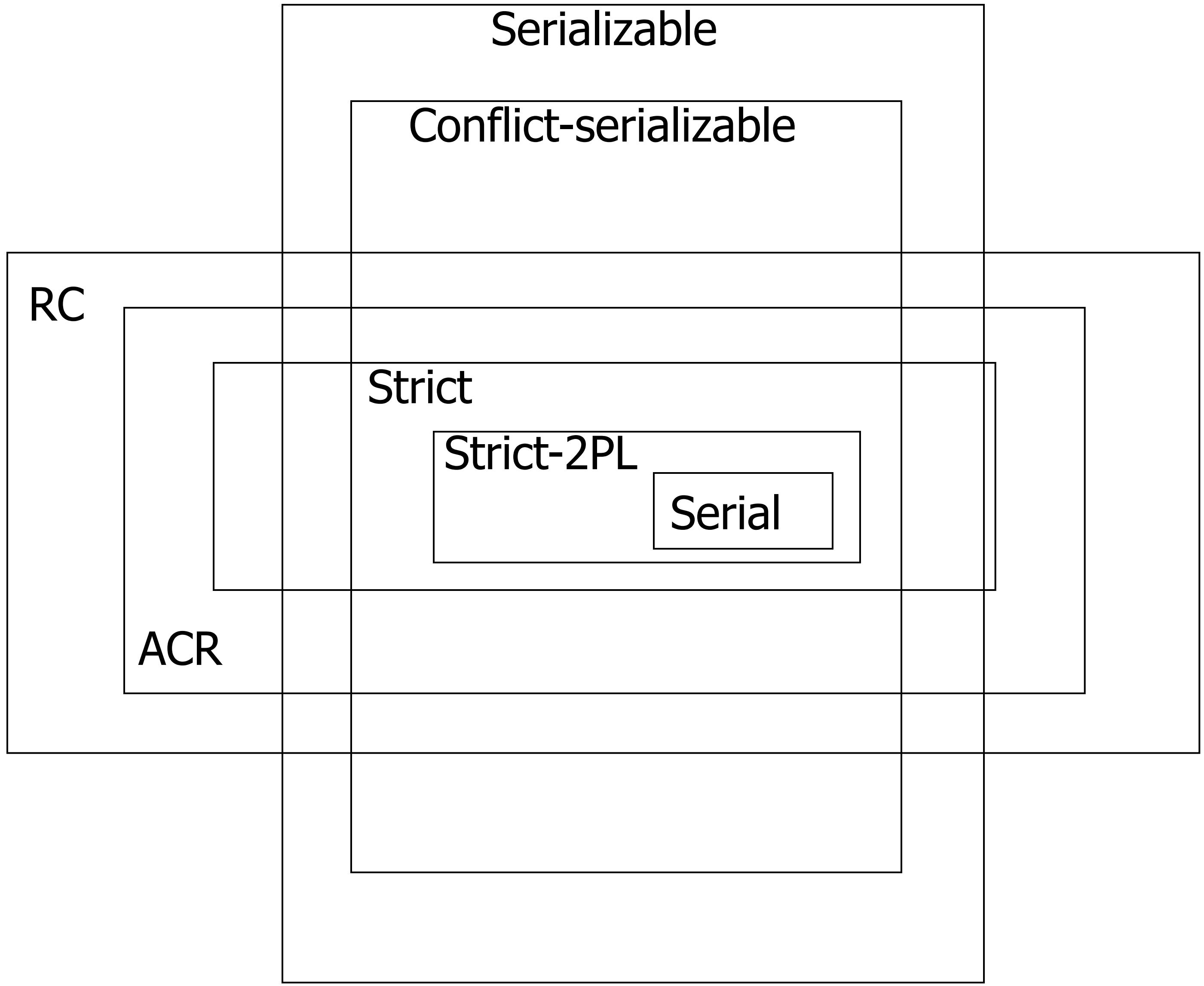
Assumes $w_2(A)$ is done without reading

- Strict:
  - ➢ $w_1(A)\ w_1(B)\ c_1\ \ w_2(A)\ r_2(B)\ \ \ c_2$

Claims

1. A conflict-serializable schedule may not be recoverable (first slide).
2. Every strict schedule is ACR, and each ACR is recoverable.
3. A strict schedule may not be serializable. Example: $r_1(a)\ w_2(a)\ w_2(b)\ c_2\ w_1(b)\ c_1$
4. Strict-2PL schedules are produced when all the locks are released at the end.
   - All strict-2PL schedules are strict as well as conflict-serializable.

RC, ACR, and Strict Schedules

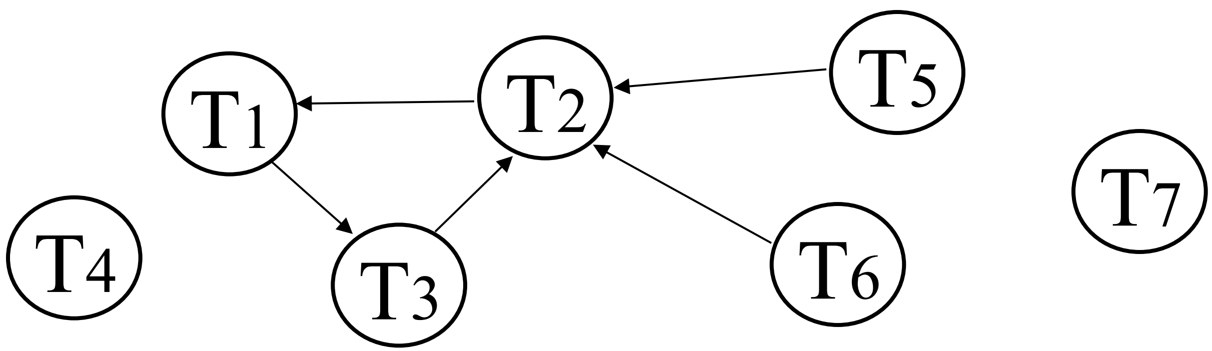


Deadlocks

- Wait-For Graph
- Prevention
  - Resource ordering
- Detection/Fixing
  - Timeout
  - Wait-die
  - Wound-wait

Tool for Prevention/Detection

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



Prevention I: Resource Ordering

- Order all elements $A_1, A_2, \ldots, A_n$
- A transaction T can lock A_i after A_j only if $i > j$

Problem: Ordered lock requests not realistic in most cases

Detection I: Timeout

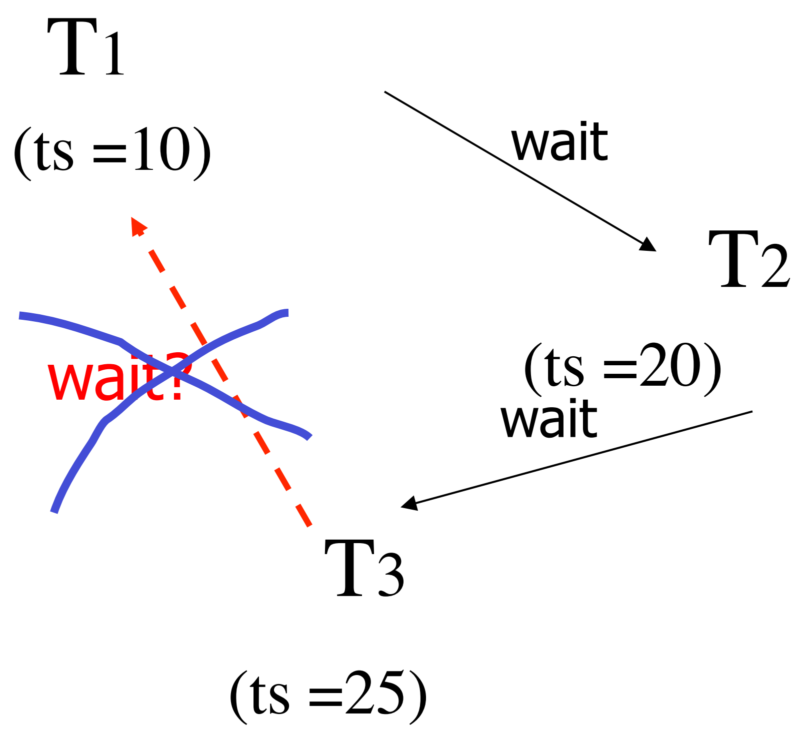
- If transaction waits more than L secs, roll it back!
- Simple scheme
- Hard to select L

Detection II: Wait-die

- Transactions given a timestamp when they arrive …. $ts(T_i)$
- T_i can only wait for T_j if it is older than T_j (i.e., $ts(T_i) < ts(T_j)$) ... else die.

- Restart with the original timestamp. Why?

Example:

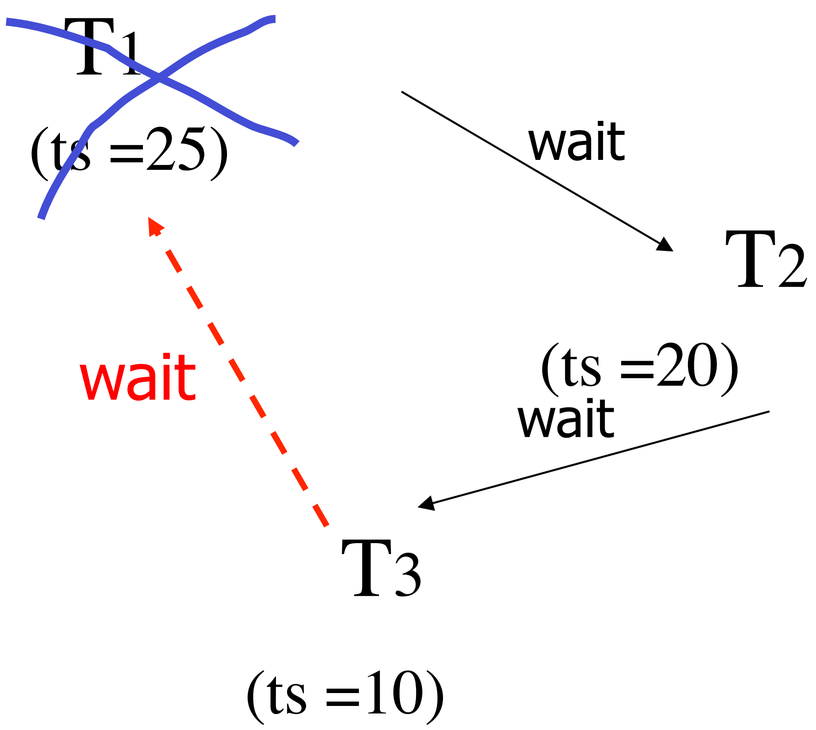


Detection III: Wound-wait

- Transactions given a timestamp when they arrive … $ts(T_i)$
- T_i wounds T_j if Ti is older than Tj (i.e., $ts(T_i) < ts(T_j)$) .. else T_i waits

"Wound": T_j rolls back and gives lock to T_i

Example:



Comparison

- Deadlock Detection: Reacts only when there is indeed a deadlock. However, they are difficult to implement.
- Wait-die vs. Wound-wait:
  - If we assume that acquisition of a lock happens early on in a transaction, then wait-die approach kills many transactions that haven't done much work. In contrast, wound-wait approach kills fewer transactions that may have already done a lot of work.