Chapter 14 More on transaction processing

Topics:

- Cascading rollback, recoverable schedule
- Deadlocks
  - Prevention
  - Detection
- View serializability
- Distributed transactions
- Long transactions (nested, compensation)

Concurrency control & recovery

Example:

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

☛ Non-Persistent Commit (Bad!)

Concurrency control & recovery

Example:

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

☛ Non-Persistent Commit (Bad!)

avoided by recoverable schedules

Concurrency control & recovery

Example:

T_j	T_i
⋮	⋮
$W_j(A)$	⋮
	$r_i(A)$
⋮	$w_i(B)$
⋮	⋮
Abort T_j	
	[Commit T_i]

☛ Cascading rollback (Bad!)

Concurrency control & recovery

Example:

T_j	T_i
⋮	⋮
$W_j(A)$	⋮
	$r_i(A)$
⋮	$w_i(B)$
⋮	
	⋮
Abort T_j	
	[Commit T_i]

☛ Cascading rollback (Bad!)

avoided by <u>avoids-cascading-rollback (ACR)</u> schedules

- Schedule is conflict serializable
- $T_j \longrightarrow T_i$

- But not recoverable

- 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

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

$a_k <_S r_i(A)$

Definition

Schedule S is recoverable if
whenever $T_j \Rightarrow_S T_i$ and $j \neq i$ and $C_i \in S$
then $C_j <_S C_i$

Note: in transactions, 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 C_i, A_i per transaction

How to achieve recoverable schedules?

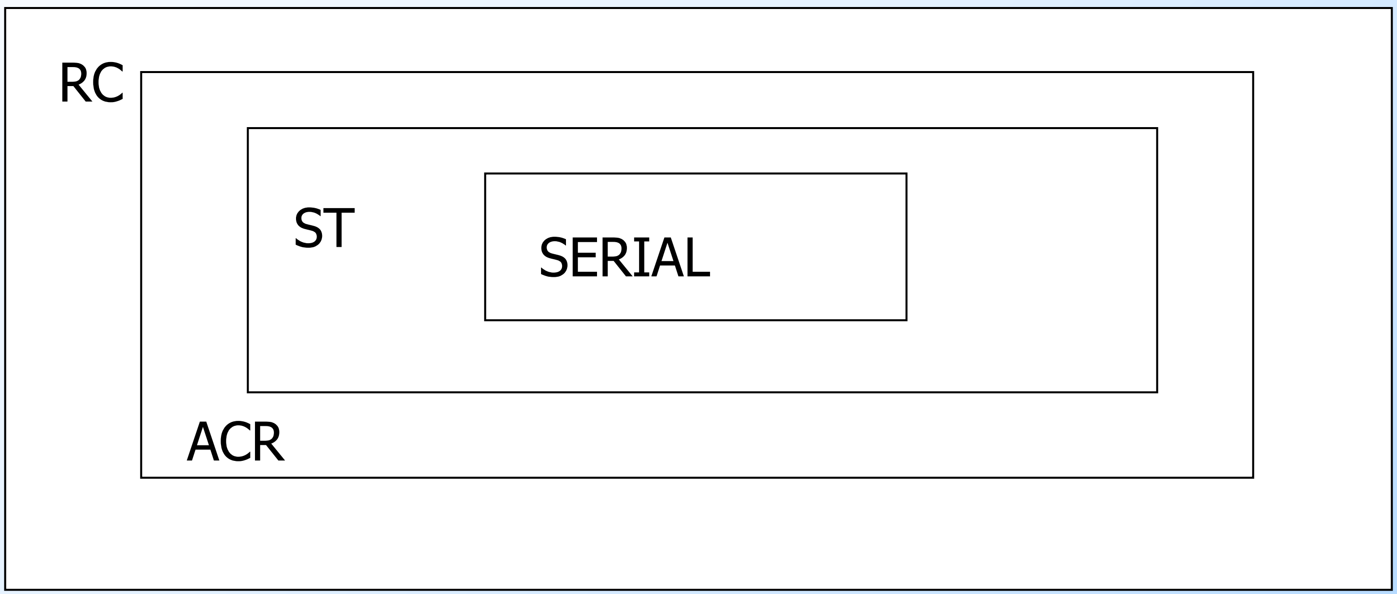
With 2PL, hold write locks to commit (strict 2PL)

T_j	T_i
$\vdots$	$\vdots$
$W_j(A)$	$\vdots$
$\vdots$	$\vdots$
C_j	$\vdots$
$u_j(A)$	
$\vdots$	$r_i(A)$

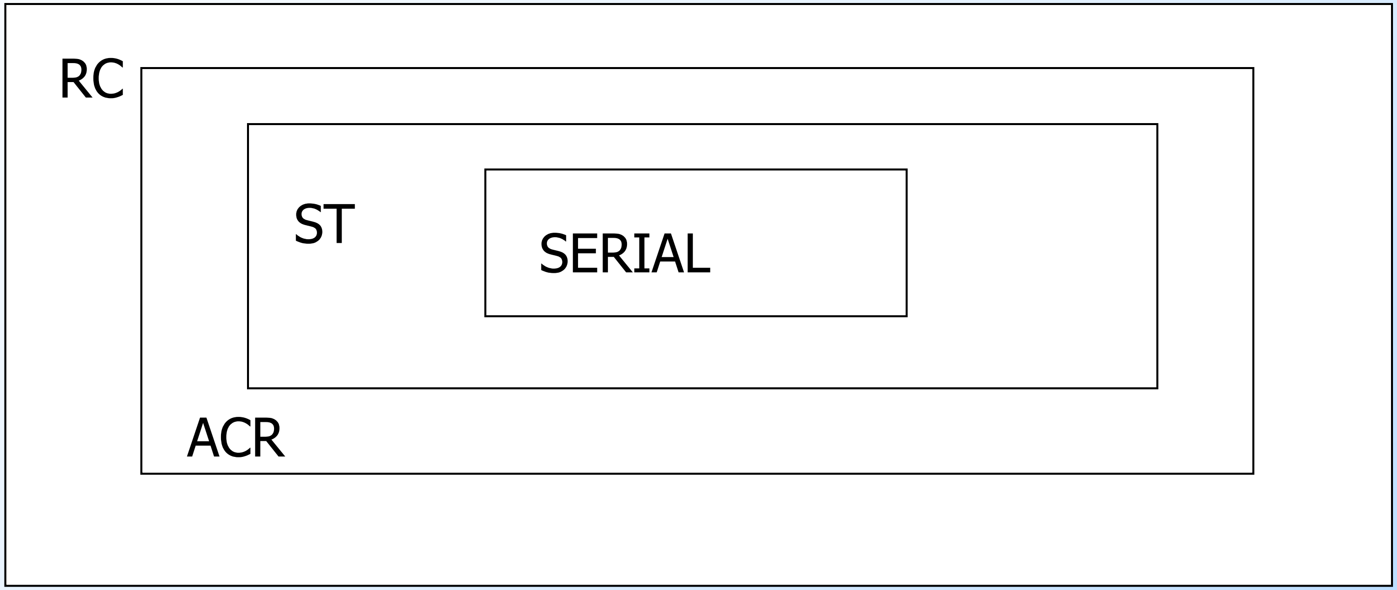
With validation, no change!

- S is <u>recoverable</u> if each transaction *commits* only after all transactions from which it read have committed.
- S <u>avoids cascading rollback</u> if each transaction may *read* only those values written by committed transactions.

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



Where are serializable schedules?



Examples

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

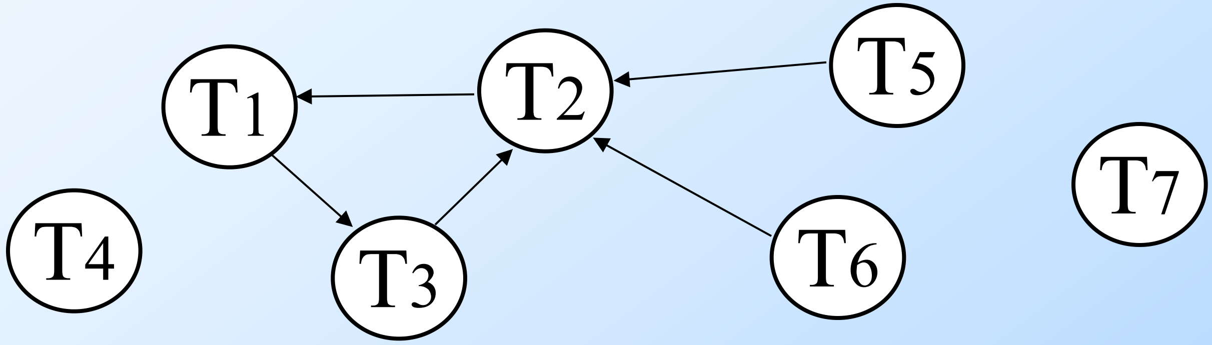
Assumes $w_2(A)$ is done without reading

Deadlocks

- Detection
  - Wait-for graph
- Prevention
  - Resource ordering
  - Timeout
  - Wait-die
  - Wound-wait

Deadlock Detection

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



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

Timeout

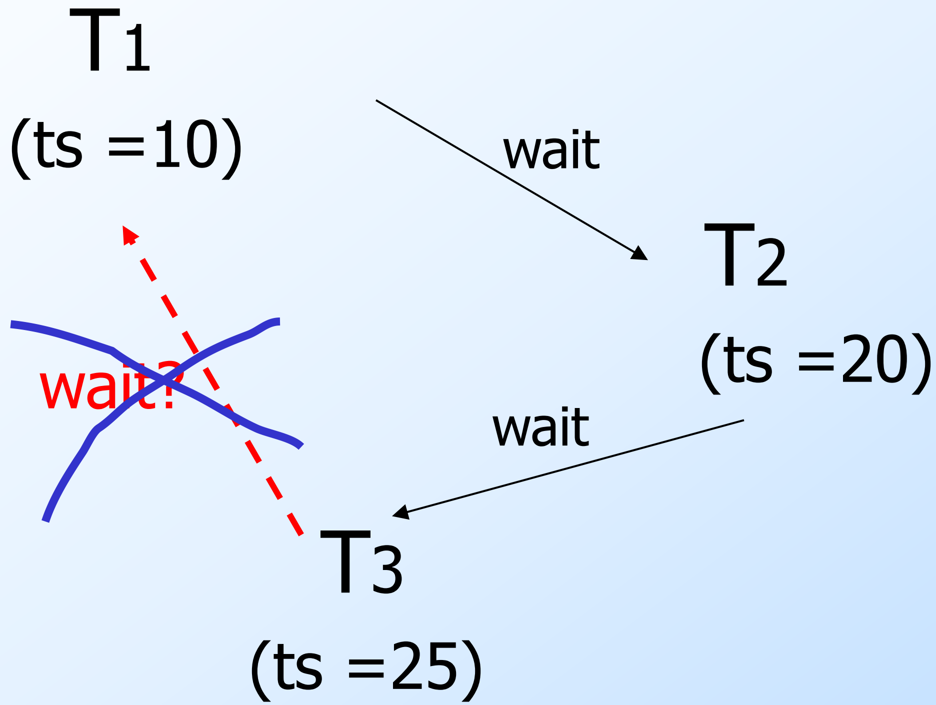
- If transaction waits more than L sec., roll it back!
- Simple scheme
- Hard to select L

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

Example:



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...

Second Example:

T_1

(ts =22)

requests A: wait for T_2 or T_3?

T_2

(ts =20)

Note: ts between
20 and 25.

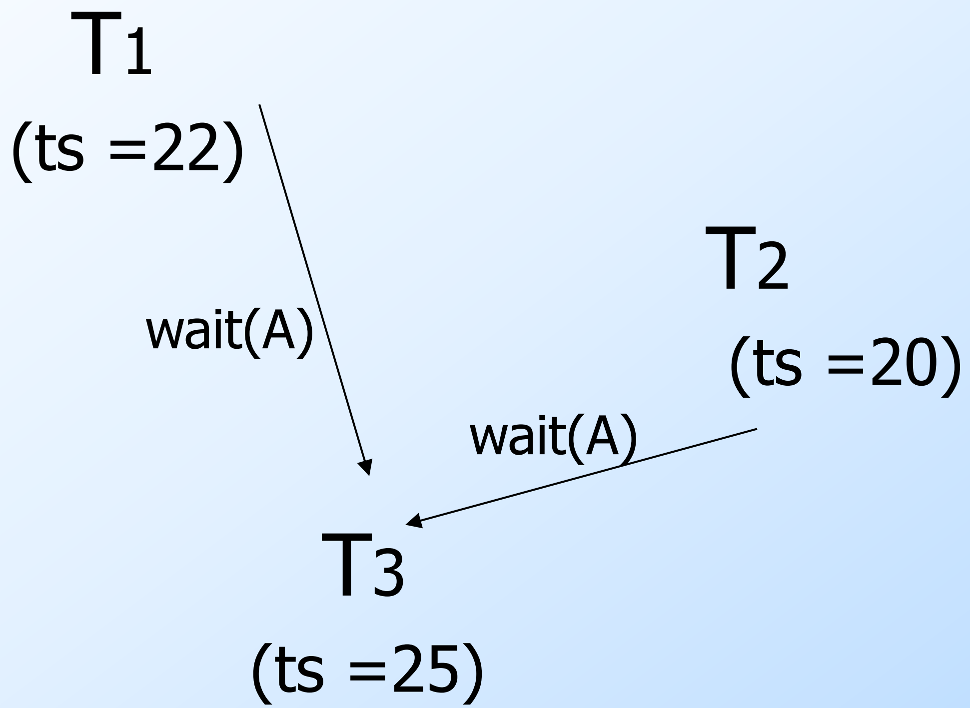
wait(A)

T_3

(ts =25)

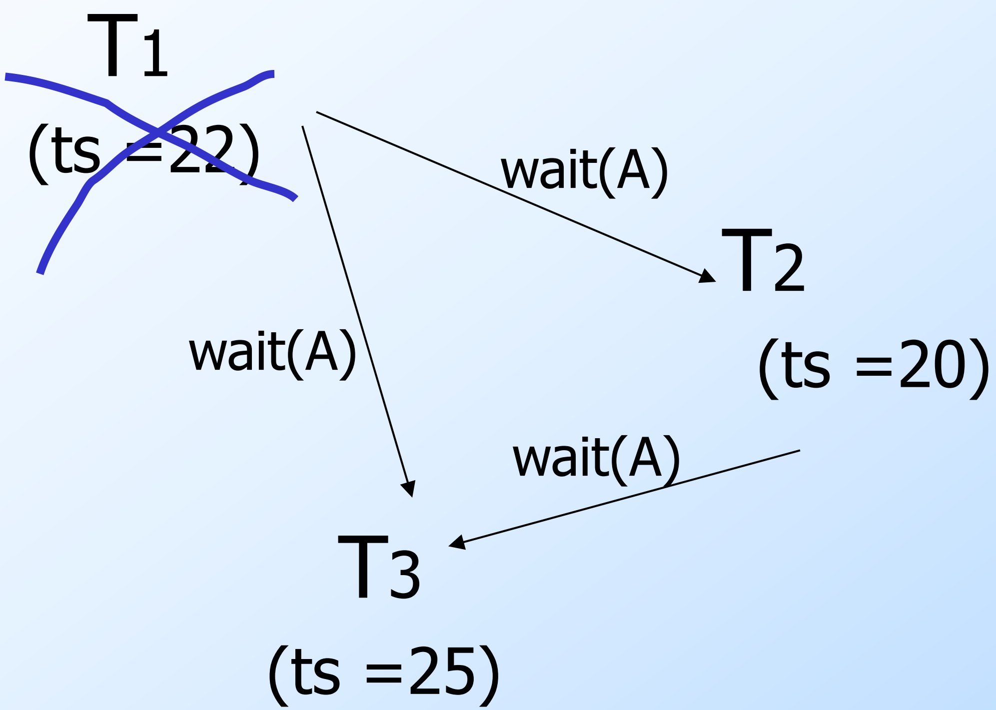
Second Example (continued):

One option: T_1 waits just for T_3, transaction holding lock.
But when T_2 gets lock, T_1 will have to die!



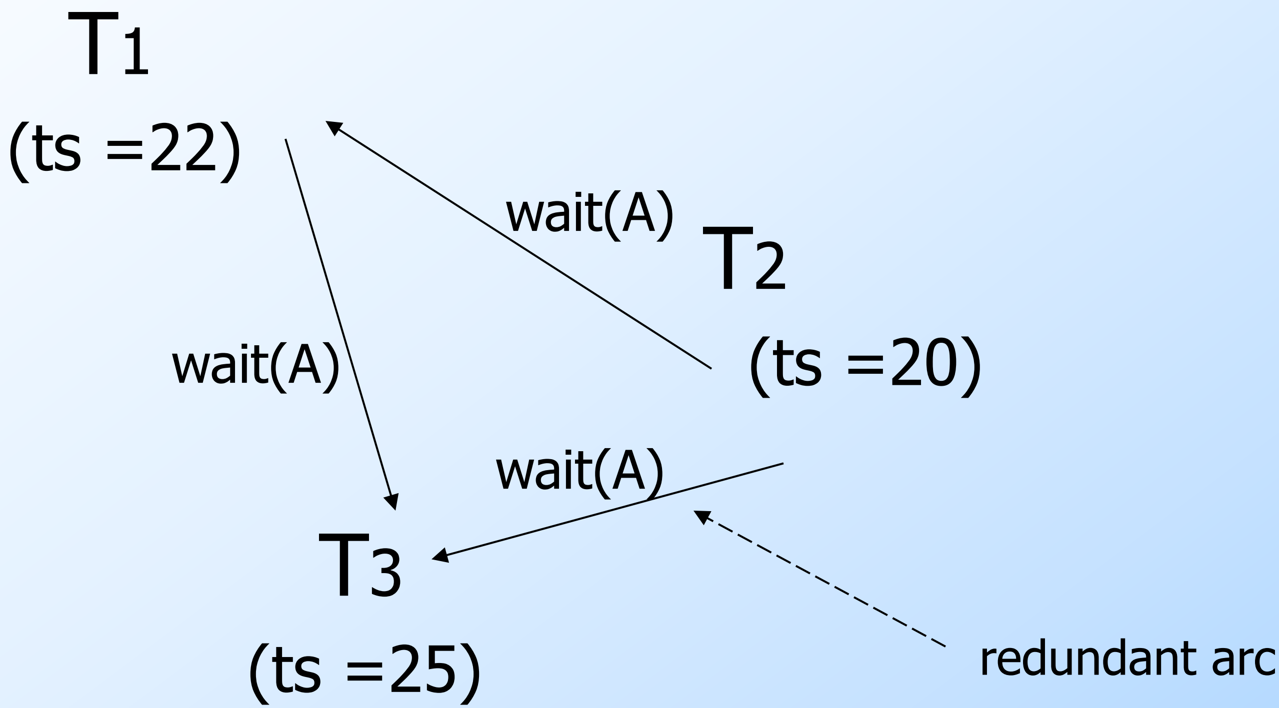
Second Example (continued):

Another option: T_1 only gets A lock after T_2, T_3 complete,
so T_1 waits for both T_2, T_3 $\Rightarrow$ T_1 dies right away!



Second Example (continued):

Yet another option: T_1 preempts T_2, so T_1 only waits for T_3; T_2 then waits for T_3 and T_1... $\Rightarrow$ T_2 may starve?



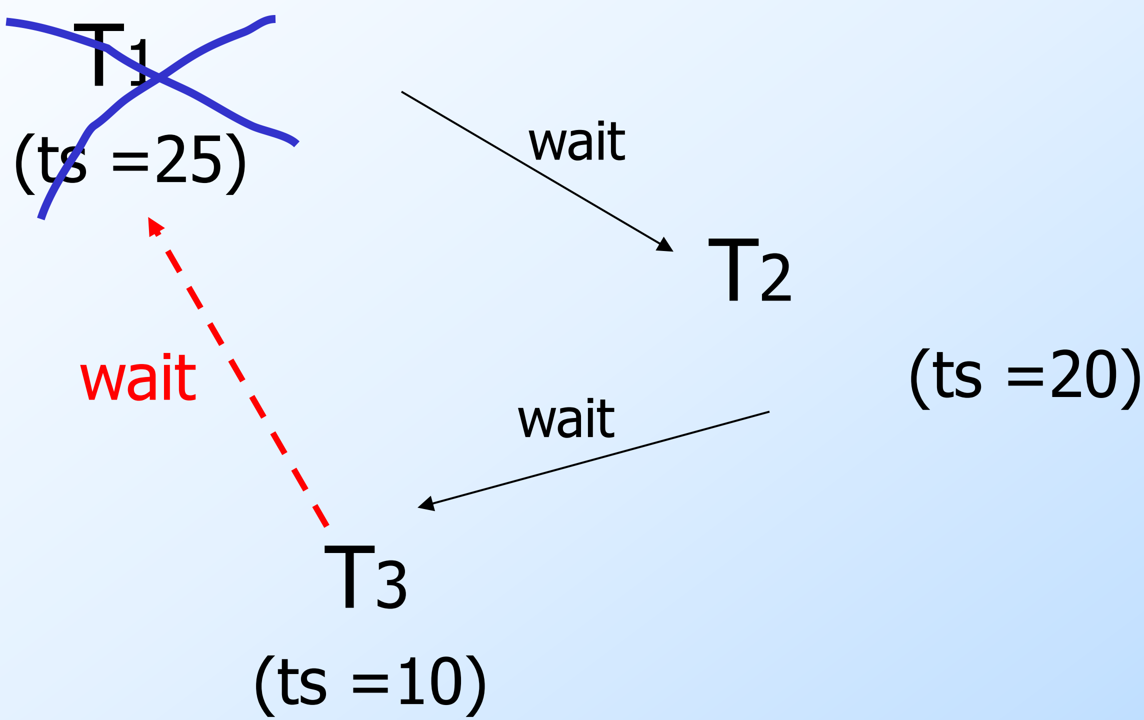
Wound-wait

- 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

Example:



Starvation with Wound-Wait

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

Second Example:

T1

(ts =15)

requests A: wait for T_2 or T_3?

Note: ts between
10 and 20.

T2

(ts =20)

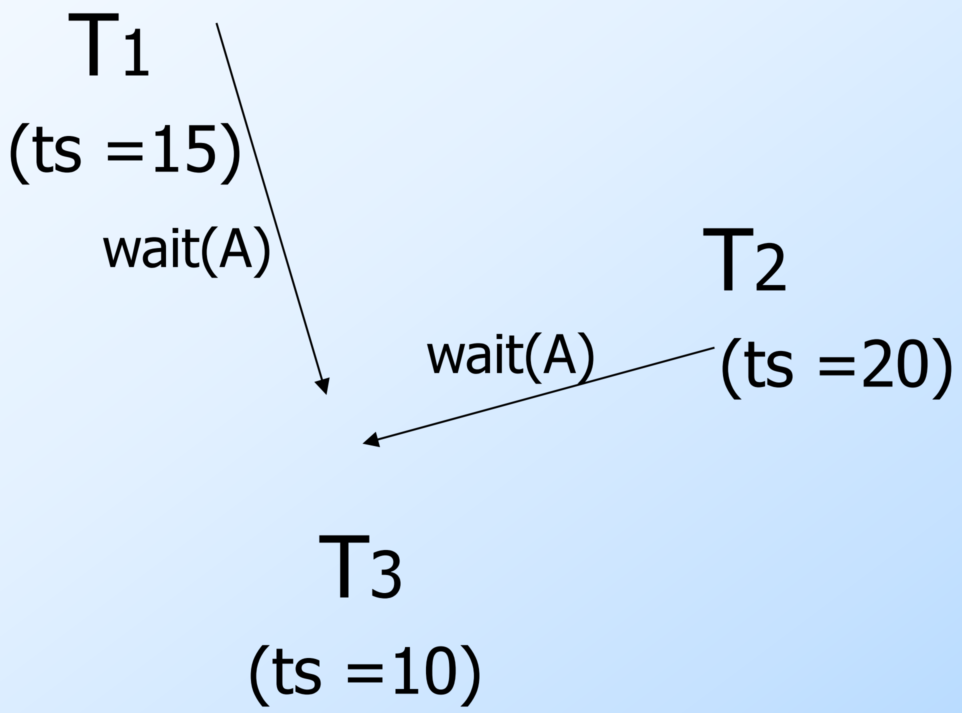
wait(A)

T3

(ts =10)

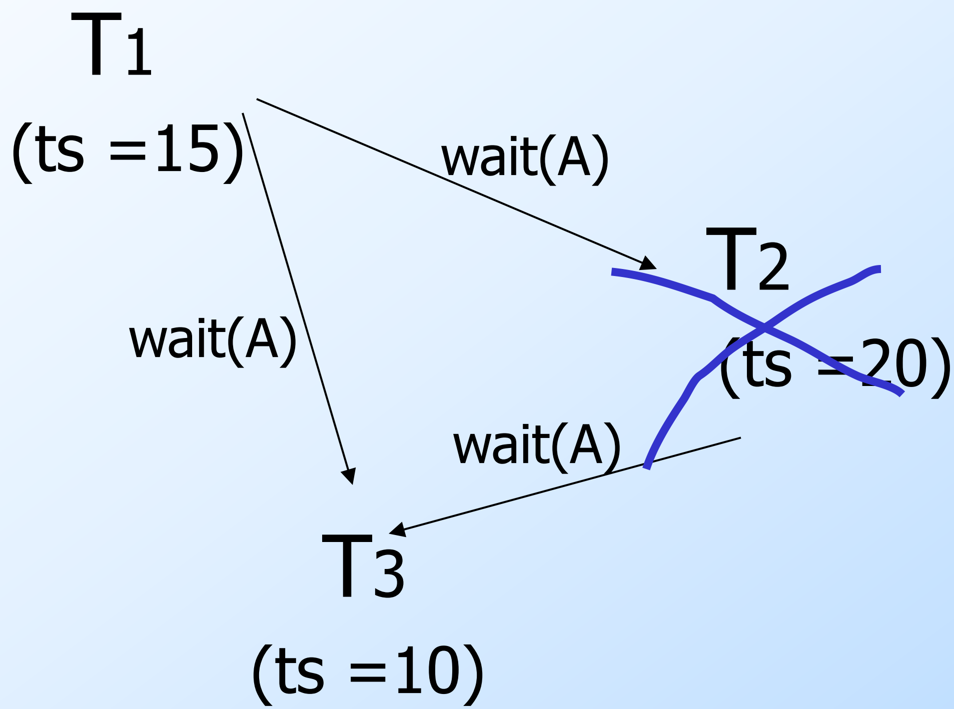
Second Example (continued):

One option: T_1 waits just for T_3, transaction holding lock.
But when T_2 gets lock, T_1 waits for T_2 and wounds T_2.



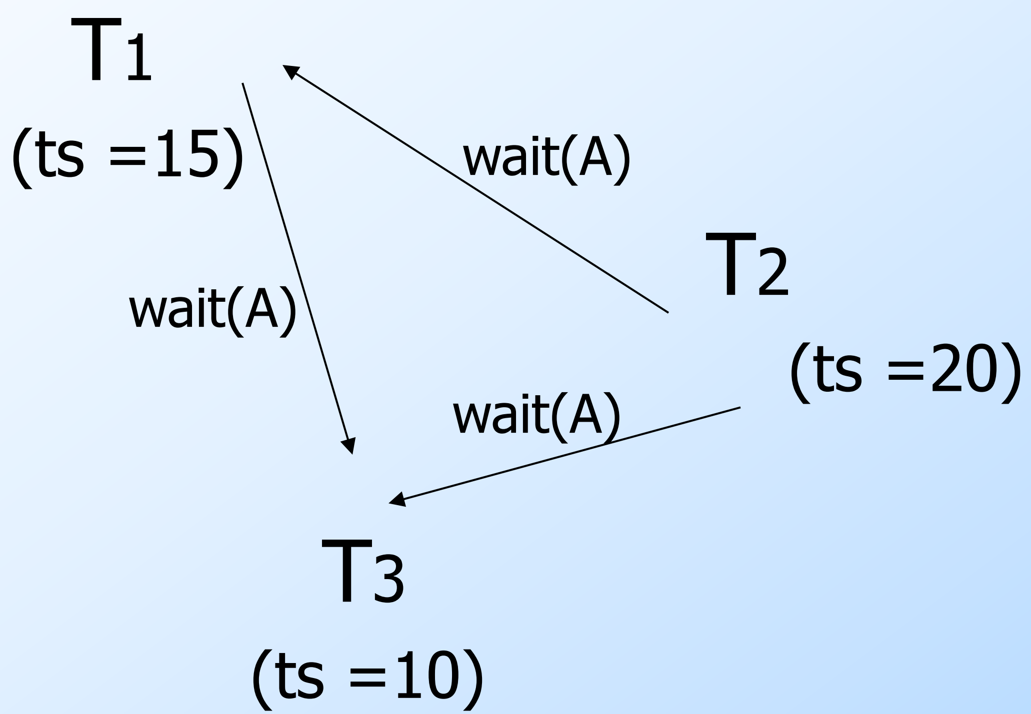
Second Example (continued):

Another option: T_1 only gets A lock after T_2, T_3 complete, so T_1 waits for both T_2, T_3 $\Rightarrow$ T_2 wounded right away!



Second Example (continued):

Yet another option: T_1 preempts T_2, so T_1 only waits for T_3; T_2 then waits for T_3 and T_1... $\Rightarrow$ T_2 is spared!



User/Program commands

Lots of variations, but in general

- Begin_work
- Commit_work
- Abort_work

Nested transactions

User program:

⋮

Begin_work;

⋮

⋮

If results_ok, then commit work

else abort_work

Nested transactions

User program:

⋮

Begin_work;

Begin_work;

⋮

If results_ok, then commit work

else {abort_work; try something else…}

⋮

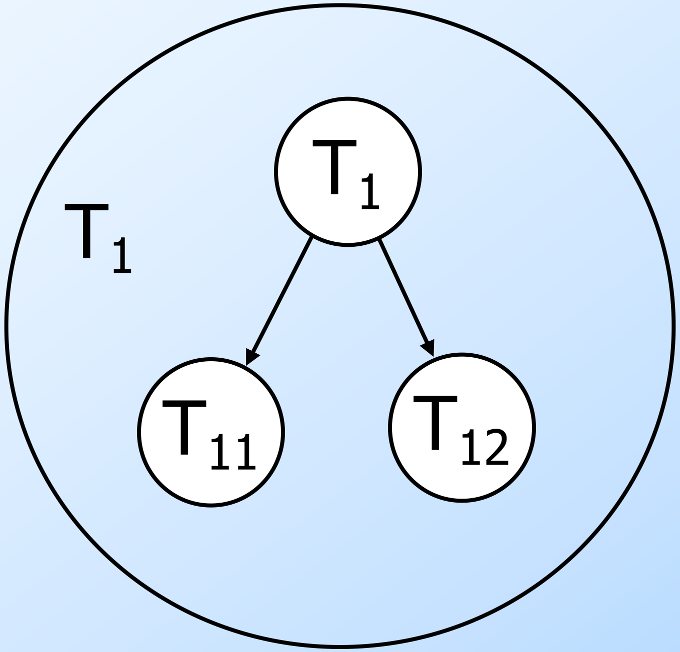
If results_ok, then commit work

else abort_work

Parallel Nested Transactions

T_1: begin-work
⋮
parallel:
T_{11}: begin_work
⋮
commit_work

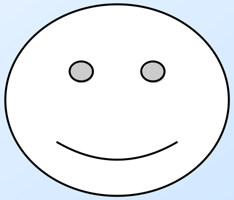
T_{12}: begin_work
⋮
commit_work
⋮
commit_work



Locking

Locking

What are we really locking?



Example:

T_i

⋮

Read record r_1

⋮

Read record r_1

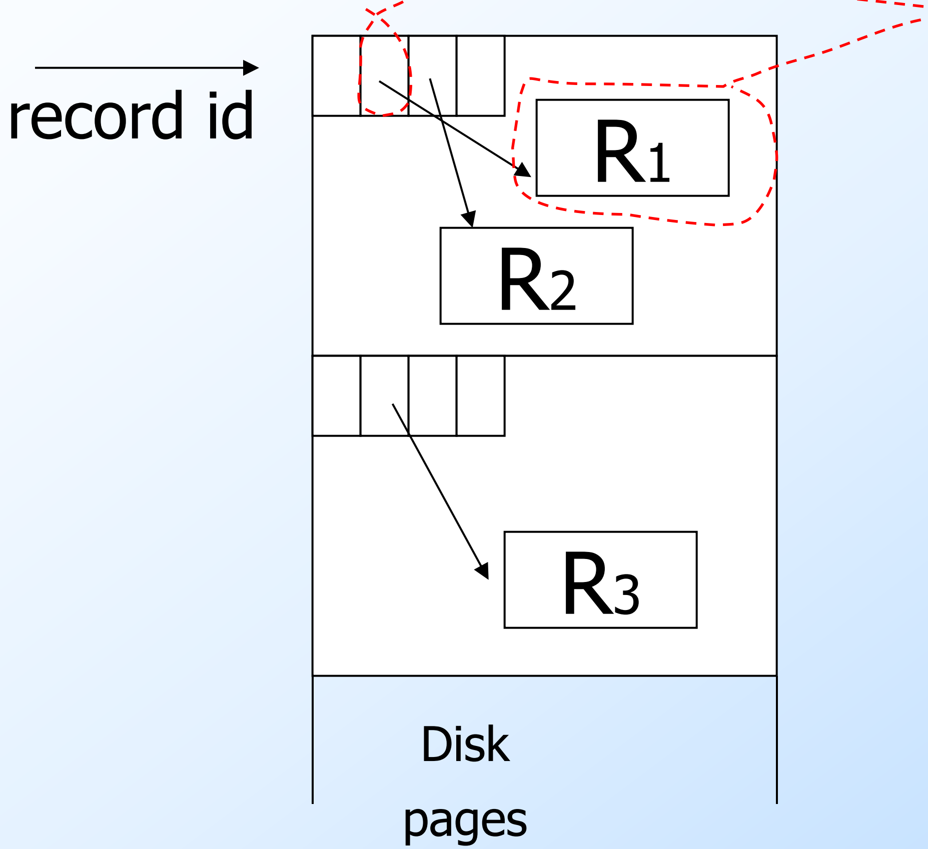
⋮

Modify record r_3

⋮

do record locking

But underneath:



If we lock all data involved in read of R1, we may prevent an update to R2 (which may require reorganization within block)

<u>Solution:</u> view DB at two levels

Top level: record actions
record locks
undo/redo actions — logical

e.g., Insert record(X,Y,Z)
Redo: insert(X,Y,Z)
Undo: delete

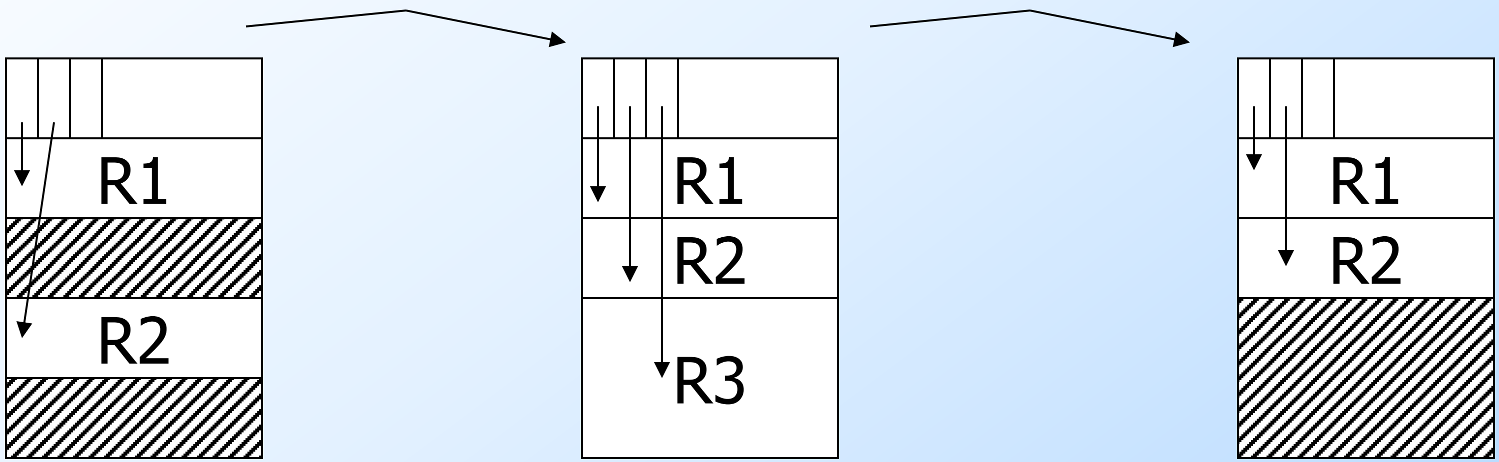
Low level: deal with physical details

latch page during action

(release at end of action)

Note: undo does not return physical DB to original state; only same logical state

e.g., Insert R3 Undo (delete R3)



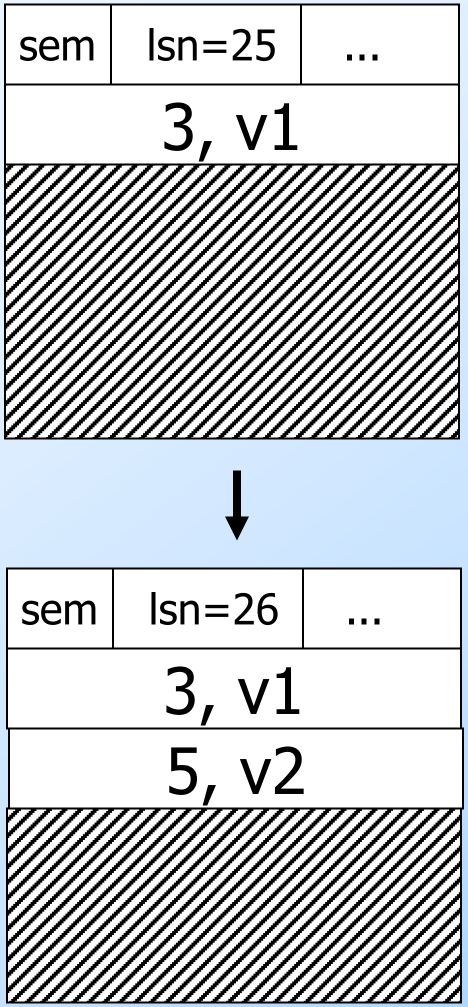
Logging Logical Actions

- Logical action typically span one block (physiological actions)
- Undo/redo log entry specifies undo/redo logical action

- Challenge: making actions idempotent
  - Example (bad): redo insert $\Rightarrow$ key inserted multiple times!

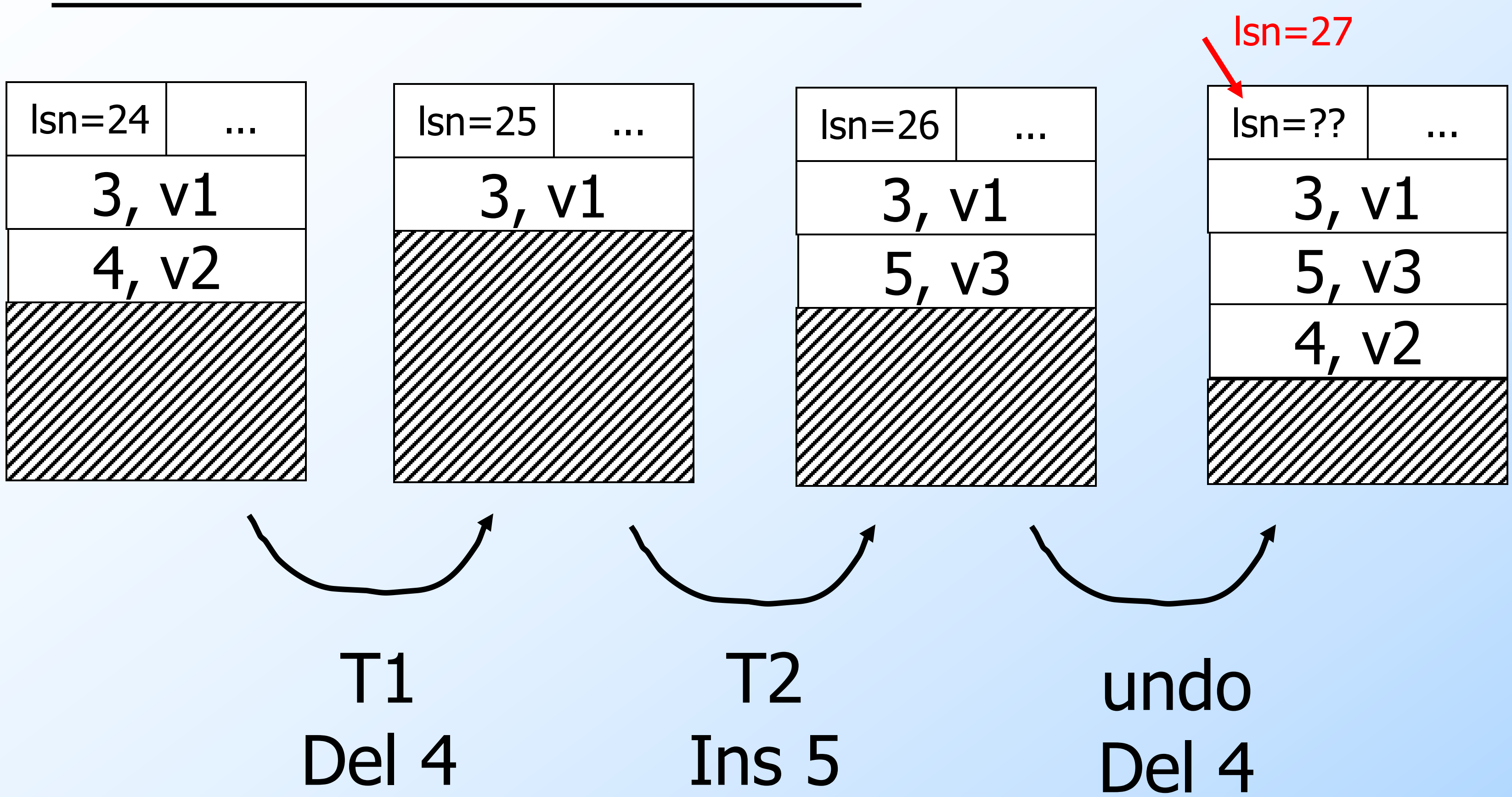
Solution: Add Log Sequence Number

Log record:
- LSN=26
- OP=insert(5,v2)
  into P
- ...



Still Have a Problem!

Make log entry
for undo



Compensation Log Records

- Log record to indicate undo (not redo) action performed
  - Note: Compensation may not return page to exactly the initial state

At Recovery: Example

Log:

…	lsn=21 T1 a1 p1	…	lsn=27 T1 a2 p2	…	lsn=35 T1 $a2^{-1}$ p2	…

What to do with p2 (during T1 rollback)?

- If lsn(p2)<27 then ... ?
- If 27 $\leq$ lsn(p2) < 35 then ... ?
- If lsn(p2) $\geq$ 35 then ... ?

Note: lsn(p2) is lsn of p copy on disk

Recovery Strategy

[1] Reconstruct state at time of crash

- Find latest valid checkpoint, *Ck*, and let *ac* be its set of active transactions
- Scan log from *Ck* to end:
  - For each log entry [lsn, page] do:
    if lsn(page) < lsn then redo action
  - If log entry is start or commit, update *ac*

Recovery Strategy

[2] Abort uncommitted transactions

- Set ac contains transactions to abort
- Scan log from end to *Ck* :
  - For each log entry (not undo) of an *ac* transaction,
    undo action (making log entry)
- For ac transactions not fully aborted, read their log entries older than *Ck* and undo their actions

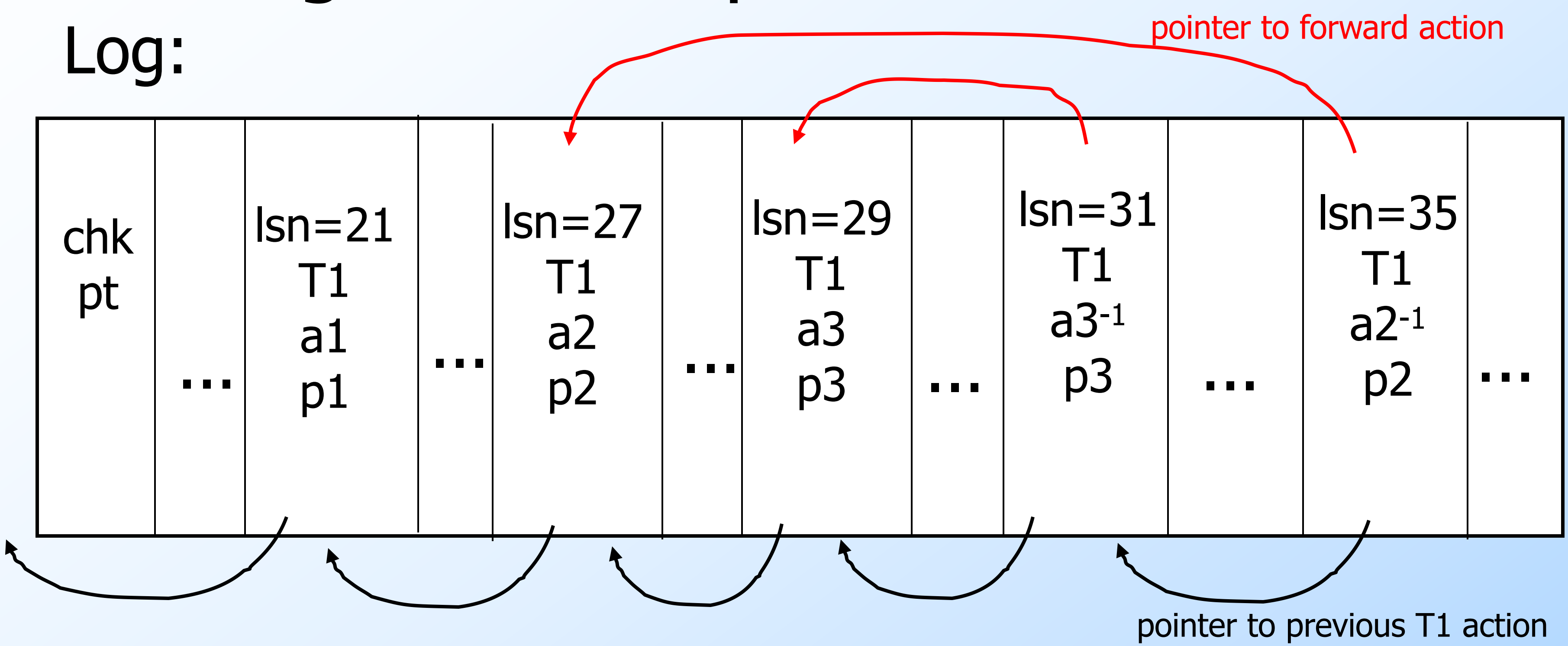
Example: What To Do After Crash

Log:

chk pt	...	lsn=21 T1 a1 p1	...	lsn=27 T1 a2 p2	...	lsn=29 T1 a3 p3	...	lsn=31 T1 $a3^{-1}$ p3	...	lsn=35 T1 $a2^{-1}$ p2	...

During Undo: Skip Undo's

Log:

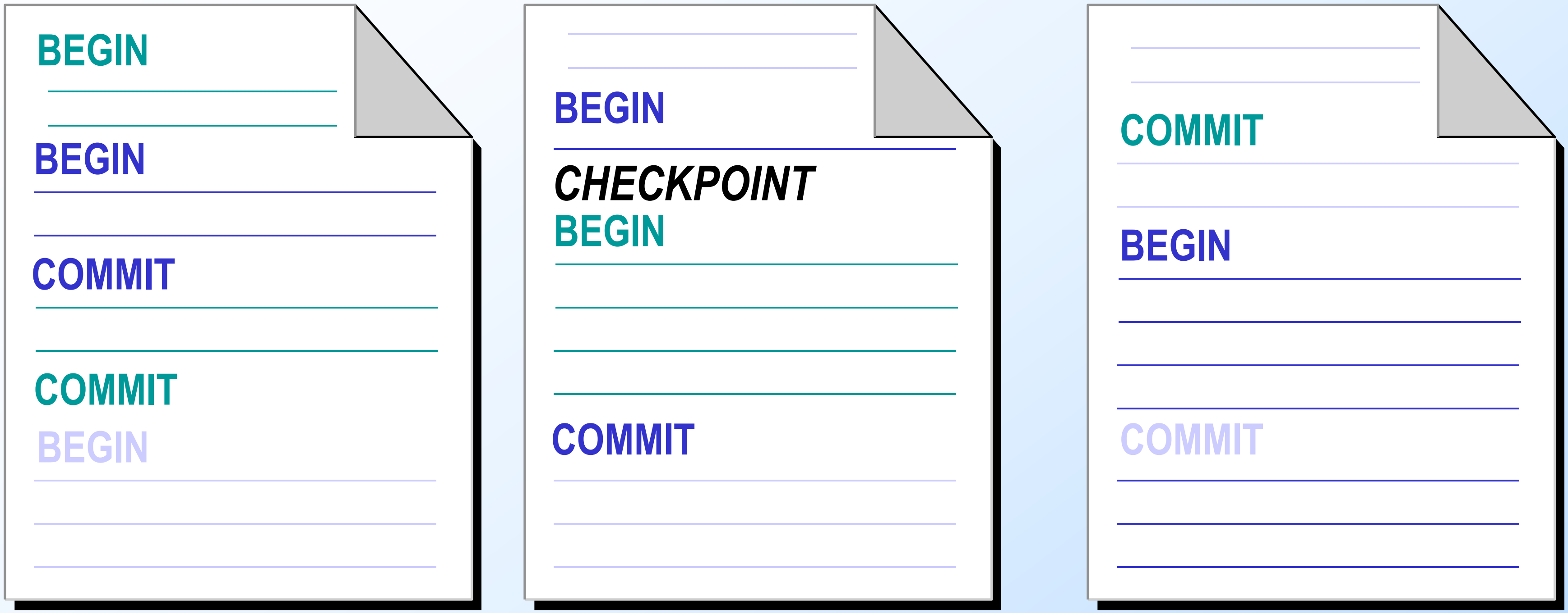


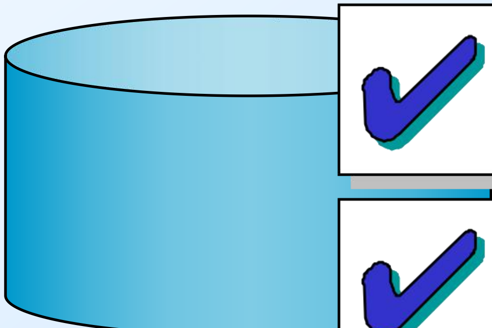
Related idea: Sagas

- Long running activity: $T_1, T_2, \ldots T_n$
- Each step/trasnaction Ti has a compensating transaction Ti-1
- Semantic atomicity: execute one of
  - $T_1, T_2, \ldots T_n$
  - $T_1, T_2, \ldots T_{n-1}\ T^{-1}_{n-1}, T^{-1}_{n-2}, \ldots T^{-1}_1$
  - $T_1, T_2, \ldots T_{n-2}\ T^{-1}_{n-2}, T^{-1}_{n-3}, \ldots T^{-1}_1$

    $\vdots$
  - $T_1,\ T^{-1}_1$
  - nothing

SQL Server Recovery Process





- Committed transactions are rolled forward and written to the database
- Uncommitted transactions are rolled back and are not written to the database

Summary

- Cascading rollback

  Recoverable schedule
- Deadlock
  - Prevention
  - Detectoin
- Nested transactions
- Multi-level view