CSE 562
Database Systems

More About Transaction Processing

Some slides are based or modified from originals by
Database Systems: The Complete Book,
Pearson Prentice Hall 2nd Edition
©2008 Garcia-Molina, Ullman, and Widom

More on Transaction Processing

Topics:

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

Concurrency Control & Recovery

Example:

Tj	Ti
⋮	⋮
Wj(A)	⋮
	ri(A)
	Commit Ti
⋮	
⋮	
Abort Tj	⋮

☛ Non-Persistent Commit (Bad!)

Concurrency Control & Recovery

Example:

Tj	Ti
⋮	⋮
Wj(A)	⋮
	ri(A)
	Commit Ti
⋮	
⋮	
Abort Tj	⋮

☛ Non-Persistent Commit (Bad!)

avoided by recoverable schedules

Concurrency Control & Recovery

Example:

Tj	Ti
⋮	⋮
Wj(A)	⋮
	ri(A)
⋮	wi(B)
⋮	
	⋮
Abort Tj	
	[Commit Ti]

☛ Cascading rollback (Bad!)

Concurrency Control & Recovery

Example:

Tj	Ti
⋮	⋮
Wj(A)	⋮
	ri(A)
⋮	wi(B)
⋮	
	⋮
Abort Tj	
	[Commit Ti]

☛ Cascading rollback (Bad!)

avoided by avoids-cascading-rollback (ACR) schedules

- Schedule is conflict serializable
- Tj ⟶ Ti

- 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:

- Ci - transaction Ti commits
- Ai - transaction Ti aborts

Back To Example:

Tj	Ti
⋮	⋮
Wj(A)	
⋮	ri(A)
	⋮
	Ci ← can we commit here?

Definition

Ti reads from Tj 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 <u>recoverable</u> 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?

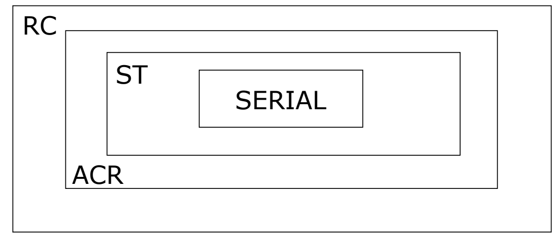
- With 2PL, hold write locks to commit (strict 2PL)

Tj	Ti
⋮	⋮
Wj(A)	⋮
⋮	⋮
Cj	
uj(A)	⋮
⋮	ri(A)

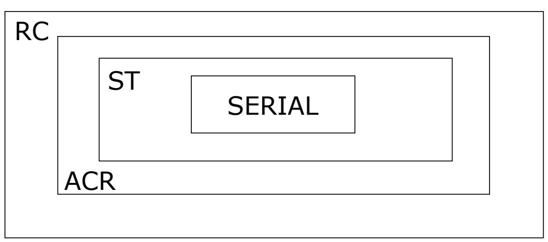
- With validation, no change!

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

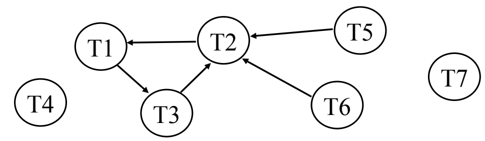
  Assumes $w_2(A)$ is done without reading
- Strict:
  - $w_1(A)\ w_1(B)\ c_1\ w_2(A)\ r_2(B)\ c_2$

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 A1, A2, …, An
- A transaction T can lock Ai after Aj only if i > j

Problem: Ordered lock requests not realistic in most cases

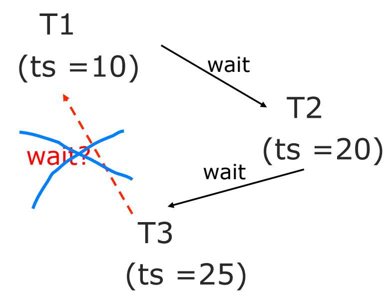
Timeout

- 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(Ti)
- Ti can only wait for Tj if ts(Ti) < ts(Tj)
  …else die

Example



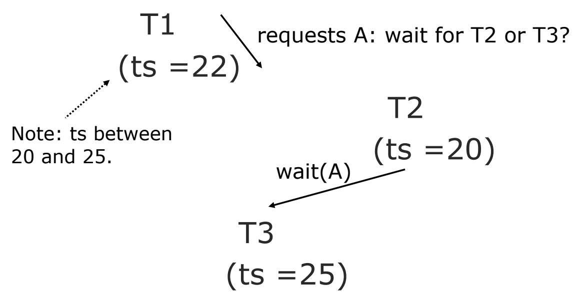
Starvation with Wait-Die

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

Starvation with Wait-Die

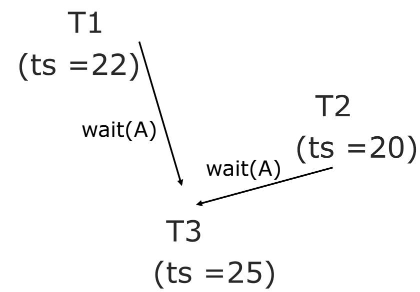
- Resubmit with original timestamp
- Guarantees no starvation
  - Transaction with oldest ts never dies
  - A transaction that dies will eventually have oldest ts and will complete…

Second Example



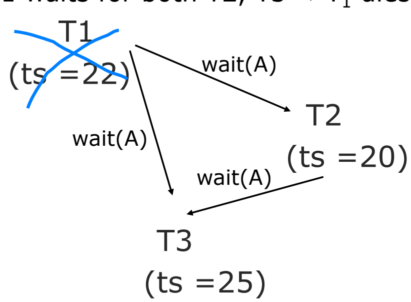
Second Example (cont'd)

One option:
T1 waits just for T3, transaction holding lock.
But when T2 gets lock, T1 will have to die!



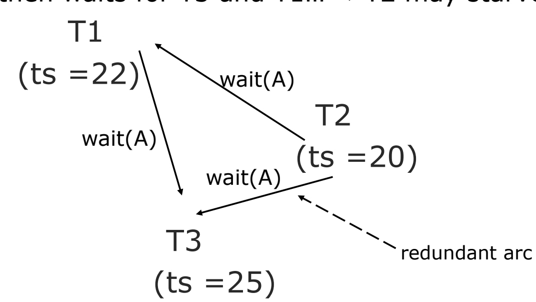
Second Example (cont'd)

Another option:
T1 only gets A lock after T2, T3 complete,
so T1 waits for both T2, T3 ⇒ T_1 dies right away!



Second Example (cont'd)

Yet another option:
T1 preempts T2, so T1 only waits for T3;
T2 then waits for T3 and T1... ⇒ T2 may starve?

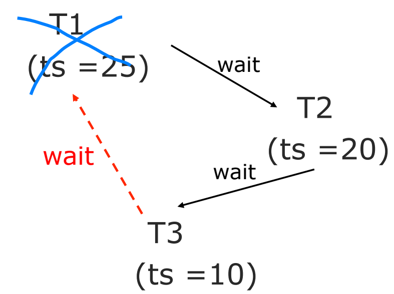


Wound-Wait

- Transactions given a timestamp when they arrive … ts(Ti)
- Ti wounds Tj if ts(Ti) < ts(Tj)
  else Ti waits

“Wound”: Tj rolls back and gives lock to Ti

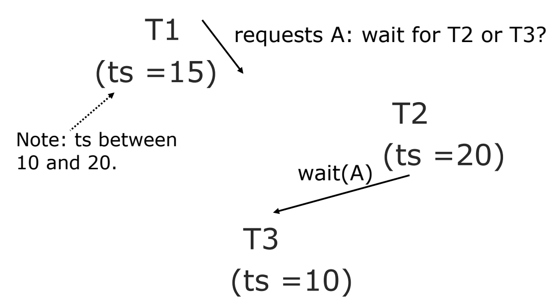
Example



Starvation with Wound-Wait

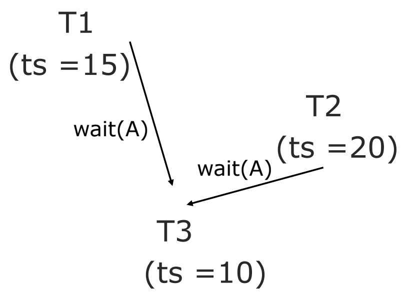
- When transaction dies, re-try later with what timestamp?
  - original timestamp
  - new timestamp (time of re-submit)

Second Example



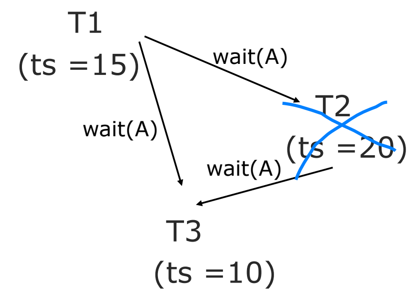
Second Example (cont'd)

One option:
T1 waits just for T3, transaction holding lock.
But when T2 gets lock, T1 waits for T2 and wounds T2.



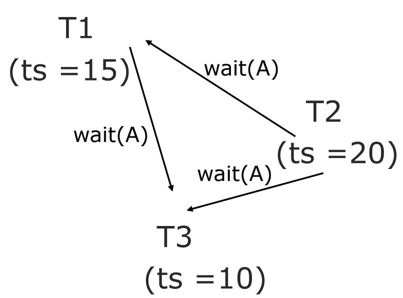
Second Example (cont'd)

Another option:
T1 only gets A lock after T2, T3 complete,
so T1 waits for both T2, T3 ⇒ T2 wounded right away!



Second Example (cont'd)

Yet another option:
T1 preempts T2, so T1 only waits for T3;
T2 then waits for T3 and T1... ⇒ T2 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

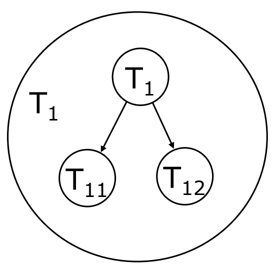
T_{12}: begin_work

⋮

commit_work

⋮

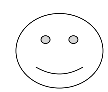
commit_work



Locking

Locking

What are we really locking?

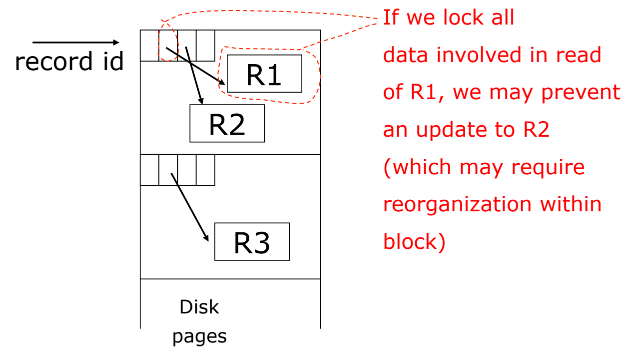


Example

Ti
⋮
Read record r1
⋮
Read record r1
⋮
Modify record r3
⋮

do record locking

But Underneath



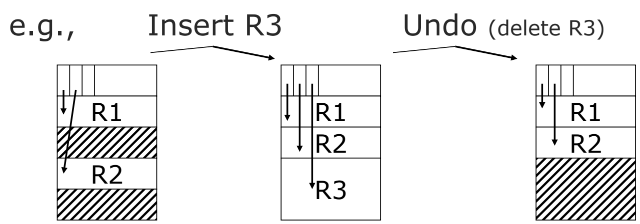
Solution: View DB At Two Levels

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

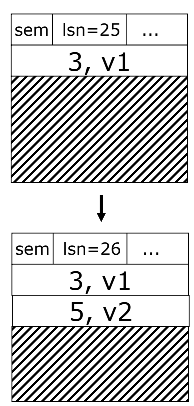


Logging Logical Actions

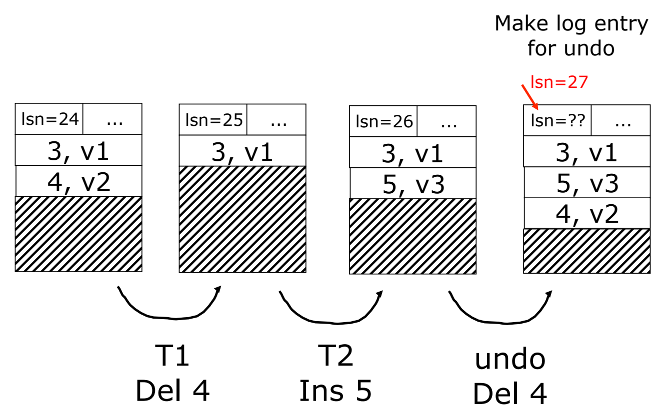
- Logical action typically span one block (physiological actions)
- Undo/redo log entry specifies undo/redo logical action
- Challenge: making actions idempotent
  - Example (bad): redo insert ⇒ key inserted multiple times!

<u>Solution:</u> Add Log Sequence Number

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



Still Have a Problem!



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 ≤ lsn(p2) < 35 then … ?
- If lsn(p2) ≥ 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

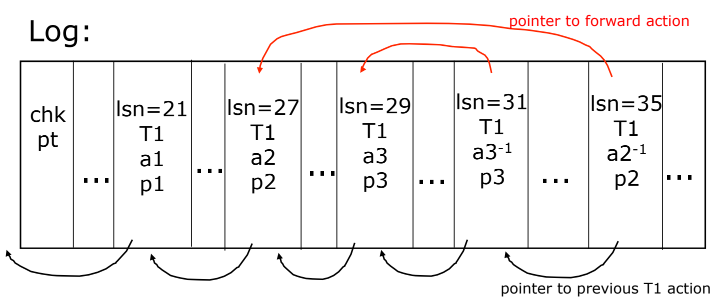
[2] Abort uncommitted transactions
- Set ac contains transactions to abort
- Scan log from end to *Ck*:
  - For each log entry (not undo) of an *ac* transaction, undo action (making log entry)
- For ac transactions not fully aborted, read their log entries older than *Ck* and undo their actions

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



Related idea: Sagas

- Long running activity: $T_1, T_2, \ldots T_n$
- Each step/transaction T_i has a compensating transaction T_{i-1}
- Semantic atomicity: execute one of
  - $T_1, T_2, \ldots T_n$
  - $T_1, T_2, \ldots T_{n-1}\ T^{-1}_{n-1}, T^{-1}_{n-2}, \ldots T^{-1}_1$
  - $T_1, T_2, \ldots T_{n-2}\ T^{-1}_{n-2}, T^{-1}_{n-3}, \ldots T^{-1}_1$
  - ⋮
  - T_1, T^{-1}_1
  - nothing

Summary

- Cascading rollback
  Recoverable schedule
- Deadlock
  - Prevention
  - Detection
- Nested transactions
- Multi-level view