

Transaction Processing

Introduction to Databases

CompSci 316 Spring 2019



DUKE
COMPUTER SCIENCE

Announcements (Thu., Apr. 11)

- Homework #4-problem 3 due Monday

Review

- ACID

- **Atomicity**: TX's are either completely done or not done at all
- **Consistency**: TX's should leave the database in a consistent state
- **Isolation**: TX's must behave as if they are executed in isolation
- **Durability**: Effects of committed TX's are resilient against failures

- SQL transactions

- Begins implicitly

- SELECT ...;

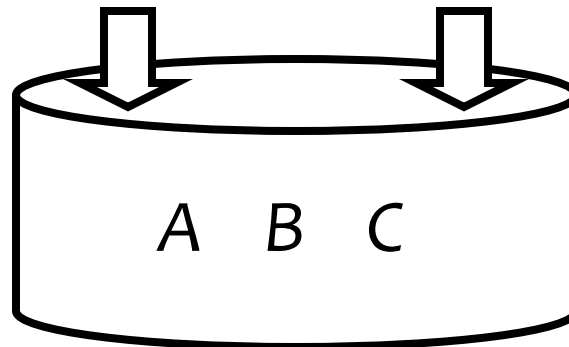
- UPDATE ...;

- ROLLBACK | COMMIT;

Concurrency control

- Goal: ensure the “I” (isolation) in ACID

T_1 :	T_2 :
read(A);	read(A);
write(A);	write(A);
read(B);	read(C);
write(B);	write(C);
commit;	commit;



Good versus bad schedules

Good!

T_1	T_2
r(A)	
w(A)	
r(B)	
w(B)	
	r(A)
	w(A)
	r(C)
	w(C)

Bad!

T_1	T_2
r(A)	
Read 400	r(A)
	Read 400
Write	w(A)
400 - 100	w(A)
	Write
	400 - 50
r(B)	r(C)
w(B)	
	w(C)

Good! (But why?)

T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
	r(C)
w(B)	
	w(C)

Serial schedule

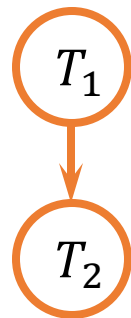
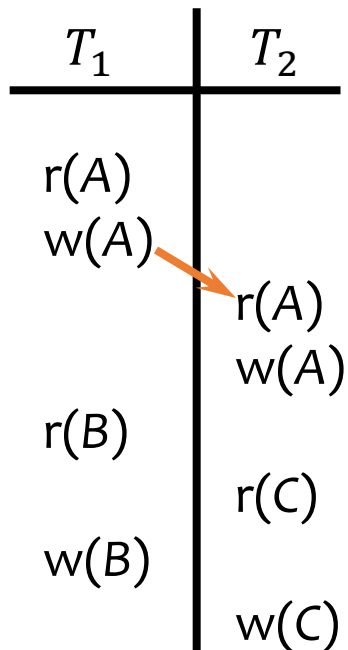
- Execute transactions in order, with **no interleaving** of operations
 - $T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C)$
 - $T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B)$
- ☞ Isolation achieved by definition!
- Problem: **no concurrency** at all
- Question: how to reorder operations to allow more concurrency

Conflicting operations

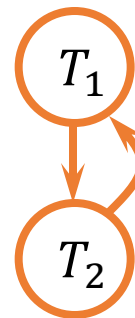
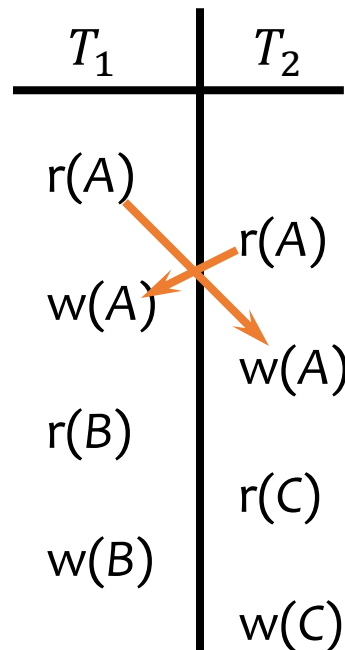
- Two operations on the **same** data item **conflict** if at least one of the operations is a write
 - $r(X)$ and $w(X)$ conflict
 - $w(X)$ and $r(X)$ conflict
 - $w(X)$ and $w(X)$ conflict
 - $r(X)$ and $r(X)$ do not conflict
 - $r/w(X)$ and $r/w(Y)$ do not conflict
- Order of conflicting operations matters
 - E.g., if $T_1.r(A)$ precedes $T_2.w(A)$, then conceptually, T_1 should precede T_2

Precedence graph

- A **node** for each transaction
- A **directed edge** from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule



Good:
no cycle



Bad:
cycle

Conflict-serializable schedule

- A schedule is **conflict-serializable** iff its precedence graph has **no cycles**
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is “good”)
 - In that serial schedule, transactions are executed in the topological order of the precedence graph
 - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions

Locking

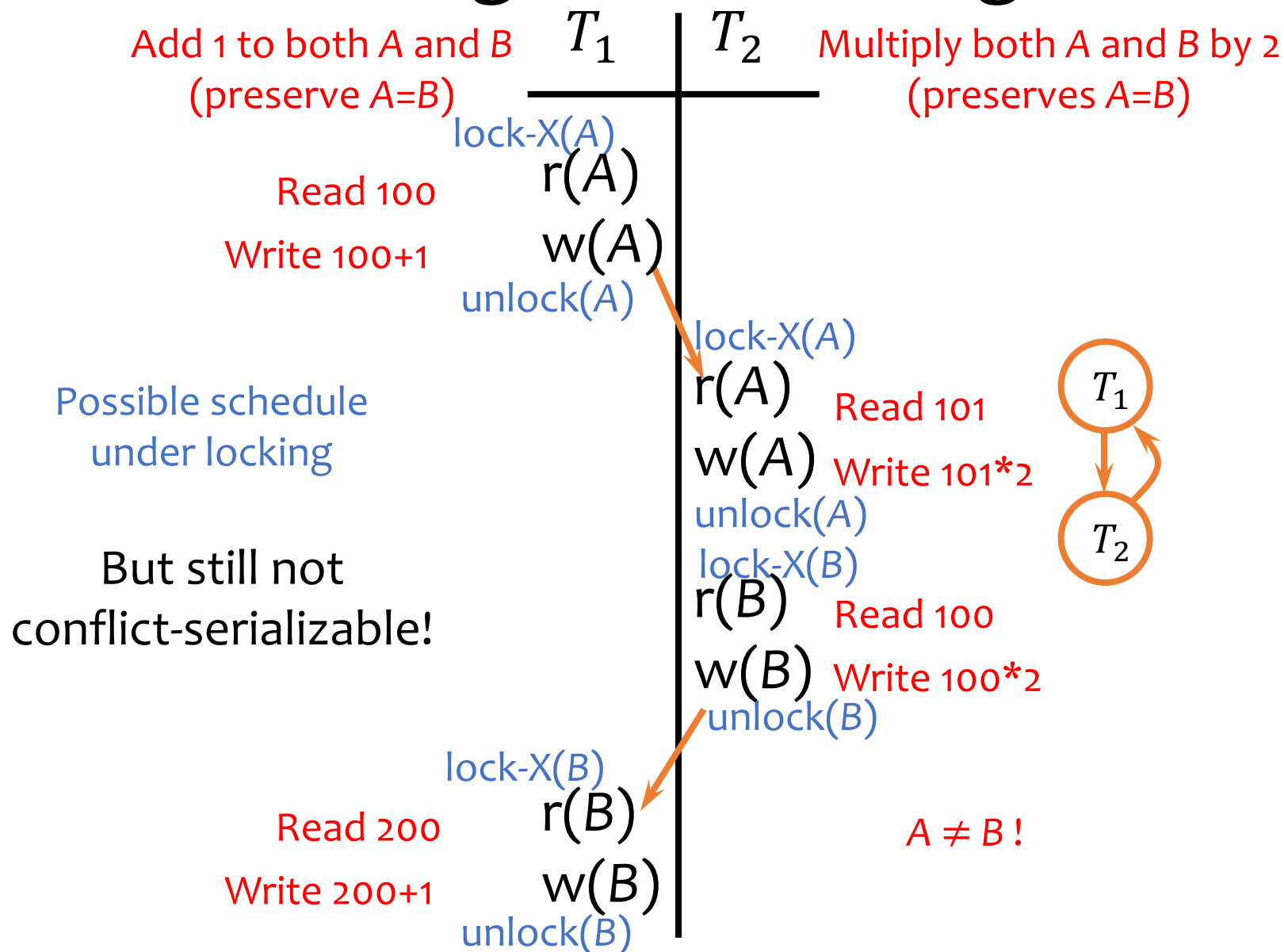
• Rules

- If a transaction wants to **read** an object, it must first request a **shared lock (S mode)** on that object
- If a transaction wants to **modify** an object, it must first request an **exclusive lock (X mode)** on that object
- Allow one exclusive lock, or multiple shared locks

		Mode of the lock requested		Grant the lock?
		S	X	
Mode of lock(s) currently held by other transactions	S	Yes	No	
	X	No	No	

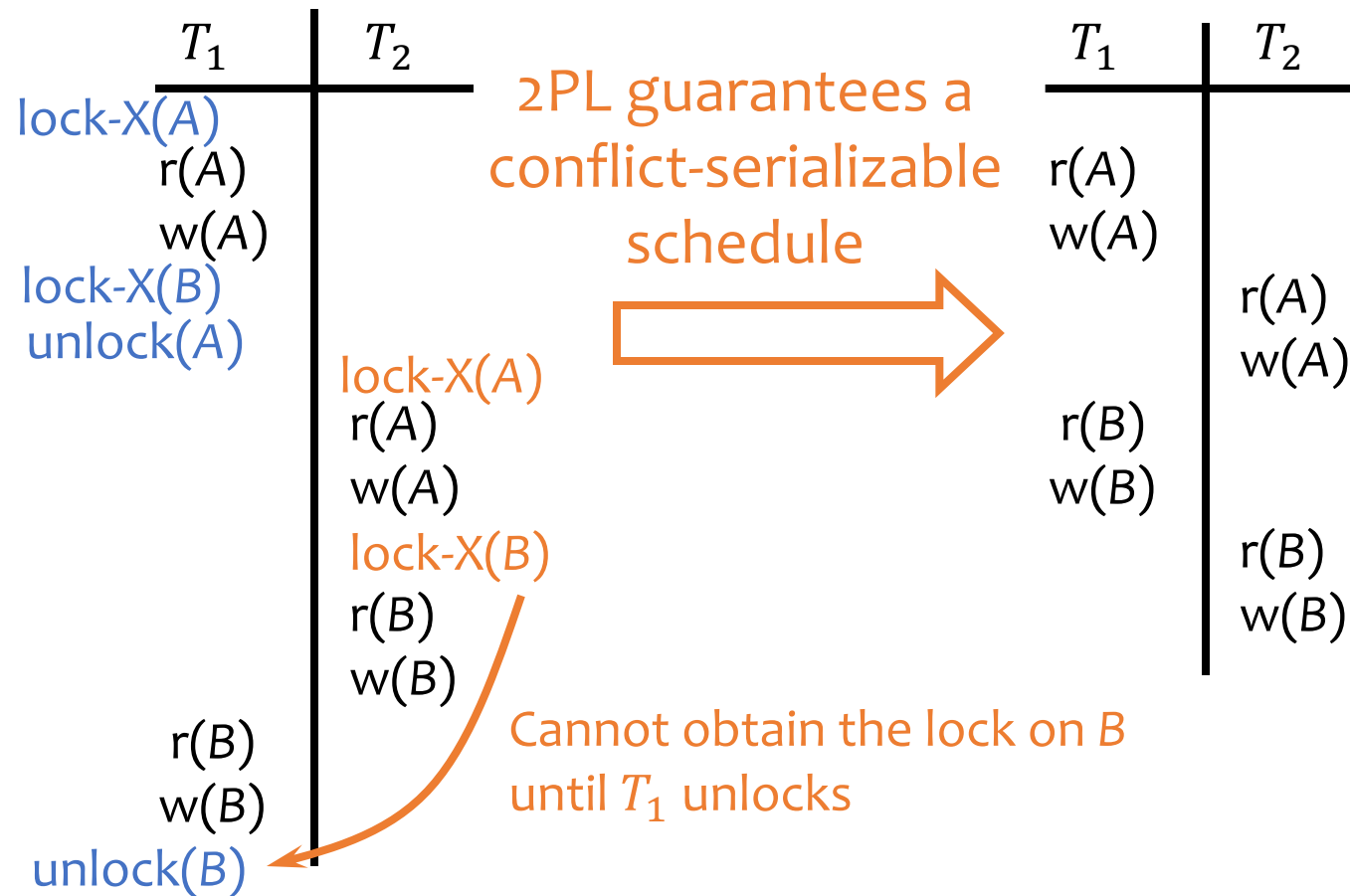
Compatibility matrix

Basic locking is not enough



Two-phase locking (2PL)

- All lock requests precede all unlock requests
 - Phase 1: obtain locks, phase 2: release locks



Remaining problems of 2PL

T_1	T_2
$r(A)$	
$w(A)$	
	$r(A)$
	$w(A)$
$r(B)$	
$w(B)$	
	$r(B)$
	$w(B)$
Abort!	

- T_2 has read uncommitted data written by T_1
 - If T_1 aborts, then T_2 must abort as well
 - **Cascading aborts** possible if other transactions have read data written by T_2
- Even worse, what if T_2 commits before T_1 ?
 - Schedule is **not recoverable** if the system crashes right after T_2 commits

Strict 2PL

- Only release locks at commit/abort time
 - A writer will block all other readers until the writer commits or aborts
- Used in many commercial DBMS
 - Oracle is a notable exception

Recovery

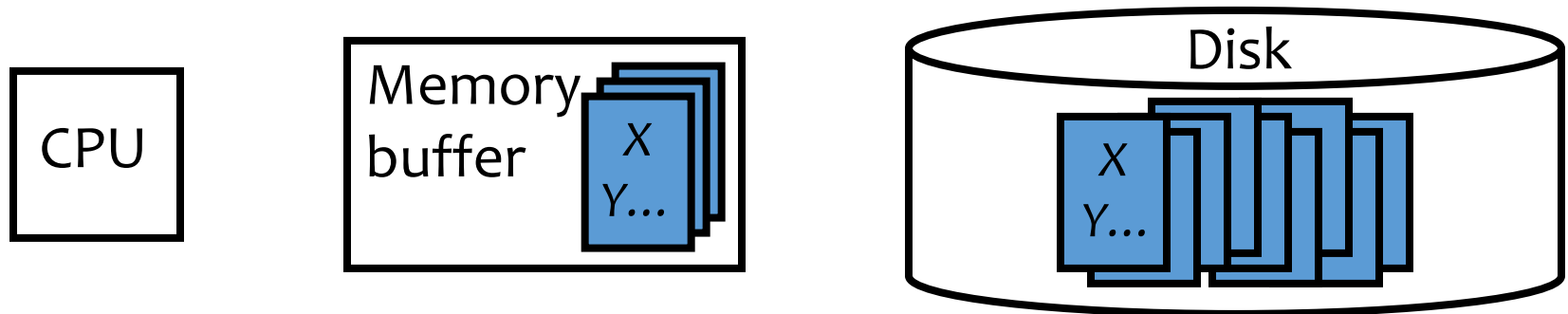
- Goal: ensure “A” (atomicity) and “D” (durability)



Execution model

To read/write X

- The disk block containing X must be first brought into memory
- X is read/written in memory
- The memory block containing X , if modified, must be written back (flushed) to disk eventually



Failures

- System crashes in the middle of a transaction T ; partial effects of T were written to disk
 - How do we undo T (atomicity)?
- System crashes right after a transaction T commits; not all effects of T were written to disk
 - How do we complete T (durability)?

Naïve approach

- **Force**: When a transaction commits, all writes of this transaction must be reflected on disk
 - Without force, if system crashes right after T commits, effects of T will be lost
 - ☞ Problem: Lots of random writes hurt performance
- **No steal**: Writes of a transaction can only be flushed to disk at commit time
 - With steal, if system crashes before T commits but after some writes of T have been flushed to disk, there is no way to undo these writes
 - ☞ Problem: Holding on to all dirty blocks requires lots of memory

Logging

- Log

- Sequence of **log records**, recording all changes made to the database
 - Written to stable storage (e.g., disk) during normal operation
 - Used in recovery
- Hey, one change turns into two—bad for performance?
 - But writes are sequential (append to the end of log)
 - Can use dedicated disk(s) to improve performance

Undo/redo logging rules

- When a transaction T_i starts, log $\langle T_i, \text{start} \rangle$
- Record values before and after each modification:
 $\langle T_i, X, \text{old_value_of_X}, \text{new_value_of_X} \rangle$
 - T_i is transaction id and X identifies the data item
- A transaction T_i is committed when its commit log record $\langle T_i, \text{commit} \rangle$ is written to disk
- **Write-ahead logging (WAL)**: Before X is modified on disk, the log record pertaining to X must be flushed
 - Without WAL, system might crash after X is modified on disk but before its log record is written to disk—no way to undo
- **No force**: A transaction can commit even if its modified memory blocks have not be written to disk (since redo information is logged)
- **Steal**: Modified memory blocks can be flushed to disk anytime (since undo information is logged)

Undo/redo logging example

T_1 (balance transfer of \$100 from A to B)

read(A, a); $a = a - 100$;

write(A, a);

read(B, b); $b = b + 100$;

write(B, b);

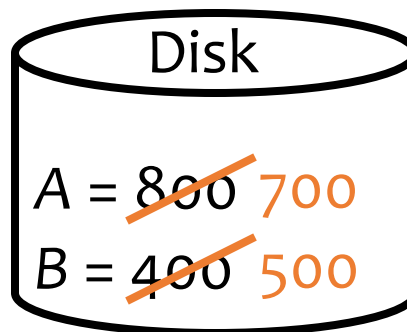
commit;

Memory buffer

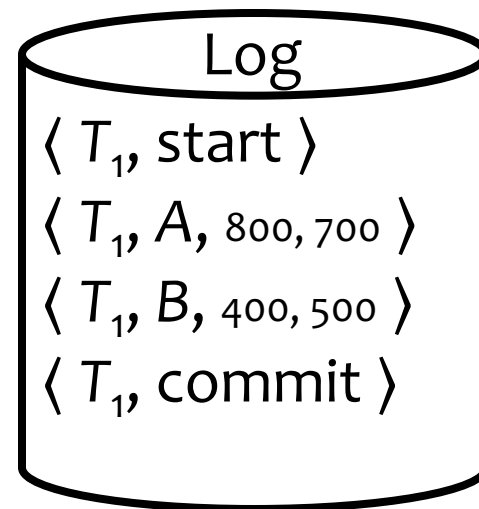
A = ~~800~~ 700

B = ~~400~~ 500

Steal: can flush
before commit



No force: can flush
after commit



No restriction (except WAL) on when memory blocks can/should be flushed

Checkpointing

- Where does recovery start?

Naïve approach:

- To checkpoint:
 - Stop accepting new transactions (lame!)
 - Finish all active transactions
 - Take a database dump
- To recover:
 - Start from last checkpoint



Fuzzy checkpointing

- Determine S , the set of (ids of) **currently active transactions**, and log $\langle \text{begin-checkpoint } S \rangle$
- Flush all blocks (dirty at the time of the checkpoint) at your leisure
- Log $\langle \text{end-checkpoint } \textit{begin-checkpoint_location} \rangle$
- Between begin and end, continue processing old and new transactions

An UNDO/REDO log with checkpointing

Log records
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

- T2 is active
- T2's new B value will be written to disk when the checkpointing begins
- During CKPT,
 - flush A to disk if it is not already there (dirty buffer)
 - flush B to disk if it is not already there (dirty buffer)

Recovery: analysis and redo phase

- Need to determine U , the set of **active transactions at time of crash**
 - Scan log backward to find the **last end-checkpoint record** and follow the pointer to find the **corresponding $\langle \text{start-checkpoint } S \rangle$**
 - Initially, let U be S
 - Scan **forward** from that start-checkpoint to end of the log
 - For a log record $\langle T, \text{start} \rangle$, add T to U
 - For a log record $\langle T, \text{commit} \mid \text{abort} \rangle$, remove T from U
 - For a log record $\langle T, X, \text{old}, \text{new} \rangle$, issue $\text{write}(X, \text{new})$
- 👉 **Basically repeats history!**

Recovery:

An UNDO/REDO log with checkpointing

Log records

<START T1>

<T1, A, 4, 5>

<START T2>

<COMMIT T1>

<T2, B, 9, 10>

→ <START CKPT(T2)>

<T2, C, 14, 15>

<START T3>

<T3, D, 19, 20>

<END CKPT>

<COMMIT T2>

<COMMIT T3>

CRASH

- T1 has committed and writes on disk
 - ignore T1
- REDO T2 and T3
- Write C = 15
- Write D = 20
- At the end U = empty, do nothing

Recovery: undo phase

- Scan log **backward**
 - Undo the effects of transactions in U
 - That is, for each log record $\langle T, X, \text{old}, \text{new} \rangle$ where T is in U , issue $\text{write}(X, \text{old})$, and log this operation too (part of the “repeating-history” paradigm)
 - Log $\langle T, \text{abort} \rangle$ when all effects of T have been undone

An optimization

- Each log record stores a pointer to the previous log record for the same transaction; follow the pointer chain during undo

Recovery:

An UNDO/REDO log with checkpointing

Log records

<START T1>

<T1, A, 4, 5>

<START T2>

<COMMIT T1>

<T2, B, 9, 10>

→ <START CKPT(T2)>

<T2, C, 14, 15>

<START T3>

<T3, D, 19, 20>

<END CKPT>

<COMMIT T2>

<COMMIT T3>

- T1 has committed and writes on disk
 - ignore T1
- T2 committed, T3 uncommitted, $U = \{T3\}$
- REDO T2 and UNDO T3
- For T2
 - set C to 15
 - not necessary to set B to 10 (before END CKPT – already on disk)
- For T3
 - reset D to 19
 - if T3 had started before START CKPT, would have had to look before START CKPT for more actions to be undone

Summary

- Concurrency control
 - Serial schedule: no interleaving
 - Conflict-serializable schedule: no cycles in the precedence graph; equivalent to a serial schedule
 - 2PL: guarantees a conflict-serializable schedule
 - Strict 2PL: also guarantees recoverability
- Recovery: undo/redo logging with fuzzy checkpointing
 - Normal operation: write-ahead logging, no force, steal
 - Recovery: first redo (forward), and then undo (backward)