

Transaction Processing

CPS 116
Introduction to Database Systems

2

Announcements (November 24)

- ❖ Homework #4 due next Tuesday
 - Help session by Dongtao next Monday 4-5pm
- ❖ Project demo period starts next week!
 - Submit your project by demo time; see project description on course website for details of what to submit
- ❖ Final exam in 2 weeks

3

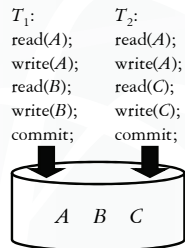
Review

- ❖ 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

4

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



Good versus bad schedules

5

Good!

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

Serial schedule

6

❖ 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

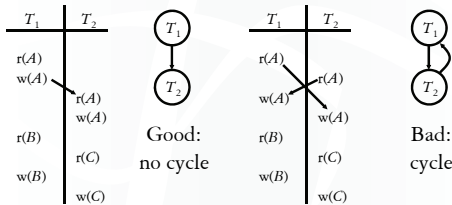
7

- ❖ 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
 - $r/w(X)$ and $r/w(Y)$ do not
- ❖ 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

8

- ❖ 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



Conflict-serializable schedule

9

- ❖ 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

10

❖ Rules

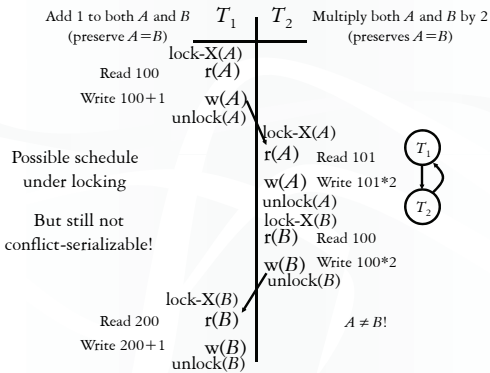
- If a transaction wants to read an object, it must first request a shared lock (S mode) on that object
- If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object
- Allow one exclusive lock, or multiple shared locks

		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

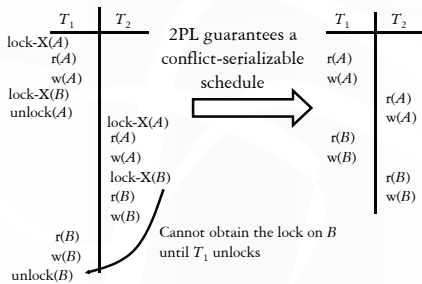
11



Two-phase locking (2PL)

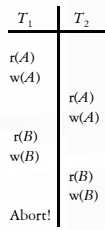
12

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



Problem of 2PL

13



- ❖ 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

Strict 2PL

14

- ❖ Only release locks at commit/abort time
 - A writer will block all other readers until the writer commits or aborts

☞ Used in most commercial DBMS (except Oracle)

Recovery

15

- ❖ Goal: ensure "A" (atomicity) and "D" (durability) in ACID
- ❖ 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

16

- ❖ 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

17

- ❖ 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:
- ❖ 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:

Logging

18

- ❖ 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?

Undo/redo logging rules

19

- ❖ Record values before and after each modification:
 $\langle T_i, X, \text{old_value_of_}X, \text{new_value_of_}X \rangle$
- ❖ 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 been 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

20

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 = 800700	
B = 400500	

Steal: can flush
before commit

No force: can flush
after commit

Disk	
A = 800700	
B = 400500	

Log	
$\langle T_1, \text{start} \rangle$	
$\langle T_1, A, 800, 700 \rangle$	
$\langle T_1, B, 400, 500 \rangle$	
$\langle T_1, \text{commit} \rangle$	

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

Checkpointing

21

- ❖ Where does recovery start?
- ❖ Naïve approach:
 - Stop accepting new transactions (lame!)
 - Finish all active transactions
 - Take a database dump
- ❖ Fuzzy checkpointing
 - Determine S , the set 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 $\langle \text{end-checkpoint } \text{begin-checkpoint_location} \rangle$
 - Between begin and end, continue processing old and new
transactions

Recovery: analysis and redo phase

22

- ❖ 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 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: undo phase

23

- ❖ 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

Summary

24

- ❖ 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)
