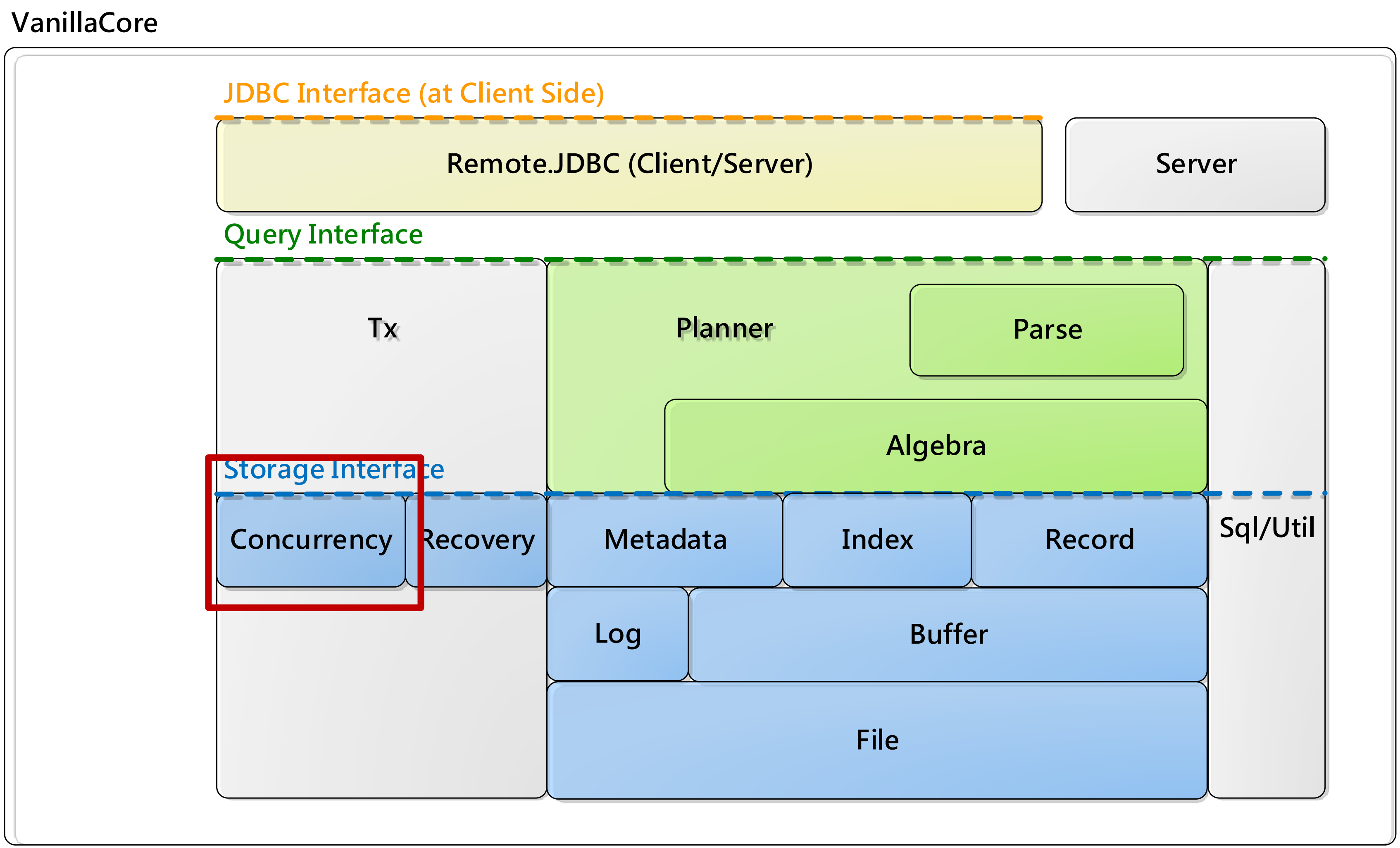
Transaction Management: Concurrency Control

DB/AI Bootcamp

2018 Summer

Datalab, CS, NTHU

Concurrency Mgr



Outline

- Schedules
- Anomalies
- Lock-based concurrency control
  - 2PL and S2PL
  - Deadlock
  - Granularity of locks
- Dynamic databases
  - Phantom
  - Isolation levels
- Meta-structures
- Concurrency manager in VanillaCore

Skipped.
Check out NTHU CS 471000 if you are interested in.

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
  - 2 Phase Locking (2PL)
  - Strict 2 Phase Locking (S2PL)
  - Deadlock

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
  - 2 Phase Locking (2PL)
  - Strict 2 Phase Locking (S2PL)
  - Deadlock

Concurrency Manager

- Ensures ***consistency*** and ***isolation***

Consistency

- ***Consistency***
  - Txs will leave the database in a consistent state
  - I.e., all integrity constraints (ICs) are meet
    - Primary and foreign key constrains
    - Non-null constrain
    - (Field) type constrain
    - …
  - Users are responsible for issuing “valid” txs

Isolation

- ***Isolation***
  - Interleaved execution of txs should have the net effect identical to executing tx in ***some*** serial order
  - T_1 and T_2 are executed concurrently, isolation gives that the net effect to be equivalent to either
    - T_1 followed by T_2 or
    - T_2 followed by T_1
  - The DBMS does ***not*** guarantee to result in ***which particular*** order

Why do we need to interleave txs?

Concurrent Txs

- Since I/O is slow, it is better to execute Tx1 and Tx2 concurrently to reduce CPU idle time

idle

Tx1	Tx2
R(A)	
CPU	
W(A)	
	R(A)
	CPU
	R(A)
	CPU
	W(B)

Tx1	Tx2
R(A)	
CPU	R(A)
W(A)	CPU
	R(A)
	CPU
	W(B)

Tx1	Tx2
R(A)	
CPU	R(A)
	CPU
	R(A)
W(A)	CPU
	W(B)

=

Tx1	Tx2
	R(A)
	CPU
	R(A)
	CPU
	W(B)
R(A)	
CPU	
W(A)	

- The concurrent result should be the same as serial execution in ***some*** order
  - Better concurrency

Concurrent Txs

- Pros:
  - Increases throughput (via CPU and I/O pipelining)
  - Shortens response time for short txs

- But operations must be interleaved correctly

Transactions and Schedules

- Before executing T_1 and T_2:
  - A = 300, B = 400

```
T1:   BEGIN  A=A+100,   B=B-100   END
T2:   BEGIN  A=1.06*A,  B=1.06*B  END
```

- Two possible execution results
  - T_1 followed by T_2
    - A = 400, B = 300 → A = 424, B = 318
  - T_2 followed by T_1
    - A = 318, B = 424 → A = 418, B = 324

Transactions and Schedules

- A ***schedule*** is a list of actions/operations from a set of transaction
- If the actions of different transactions are not interleaved, we call this schedule a ***serial schedule***

T1:	A=A+100,	B=B-100		
T2:			A=1.06*A,	B=1.06*B

Transactions and Schedules

- Equivalent schedules
  - The effect of executing the first schedule is identical to the effect of executing the second schedule
- ***Serializable schedule***
  - A schedule that is equivalent to some serial execution of the transactions

Transactions and Schedules

- A possible interleaving schedule

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

  – Result: A = 424, B = 318

  – A serializable schedule

    - Equivalent to T_1 followed by T_2

T1:	A=A+100,	B=B-100		
T2:			A=1.06*A,	B=1.06*B

Transactions and Schedules

- How about this schedule?

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

  - Result: ***A = 424***, ***B = 324***
  - A non-serializable schedule
  - Violates the isolation requirement

Goal

- Interleave operations while making sure the schedules are serializable
- How?

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
  - 2 Phase Locking (2PL)
  - Strict 2 Phase Locking (S2PL)
  - Deadlock

Simplified Notation

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- Can be simplified to:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

- Here, we care about operations, not values

Anomalies

- Weird situations that would happen when interleaving operations
  - But not in serial schedules
- Mainly due to the ***conflicting*** operations

Conflict Operations

- Two operations on the same object are ***conflict*** if they are operated by different txs and at least one of these operations is a write

T1:	R(A), W(A)		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

Types

- Write-read conflict
- Read-write conflict
- Write-write conflict

- Read-read conflict?
  - No anomaly

Anomalies due to Write-Read Conflict

- Reading uncommitted data
  - *Dirty reads*

T1:	R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)	

- A ***unrecoverable schedule***

T1:	R(A), W(A),	R(B), W(B), Abort
T2:	R(A), W(A), C	

  - T1 cannot abort!
  - ***Cascading aborts*** if T2 completes after T1 aborts

Anomalies due to Read-Write Conflict

- ***Unreadable reads***:
  - T_1: $if\ (A > 0)\ A\ =\ A - 1;$
  - T_2: $if\ (A > 0)\ A\ =\ A - 1;$
  - IC on A: cannot be negative

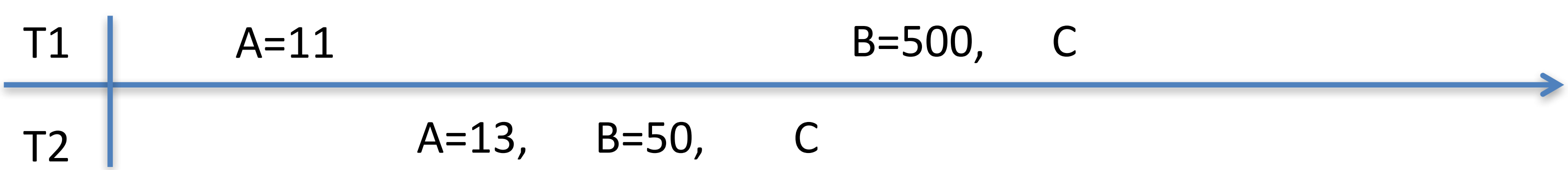
T1:	R(A),		R(A), W(A), C	
T2:		R(A), W(A), C		

T1	A=1				**A=0**,	A=-1,	C
T2		A=1,	A=0,	C			

Anomalies due to Write-Write Conflict

- ***Lost updates***:
  - $T_1: A = A + 1; B = B * 10;$
  - $T_2: A = A + 2; B = B * 5;$
  - Start with A=10, B=10

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

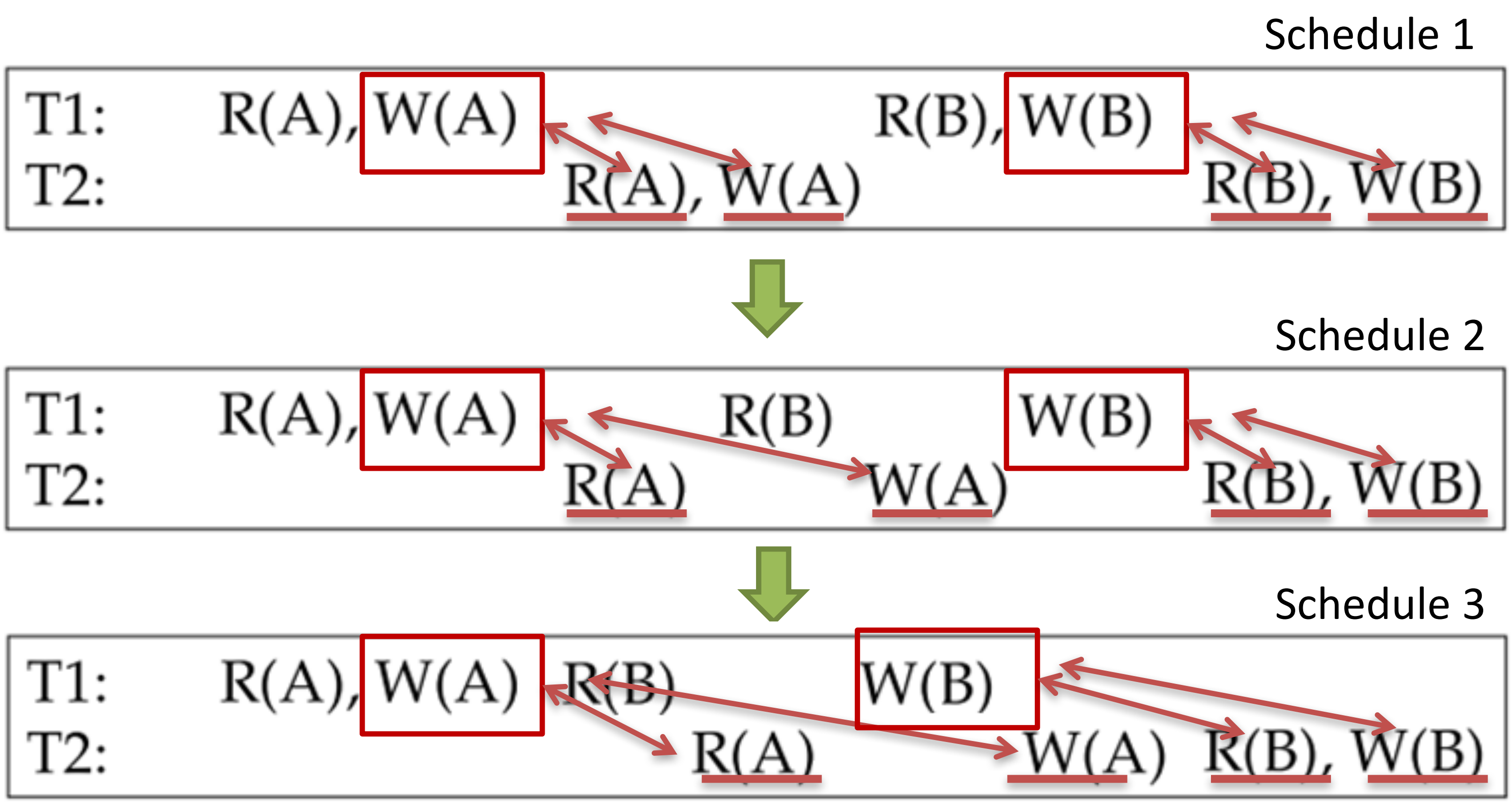


Avoiding Anomalies

- Idea:
- Perform all conflicting actions between T1 and T2 ***in the same order*** (either T1's before T2's or T2's before T1's)
- I.e., to ensure ***conflict serializability***

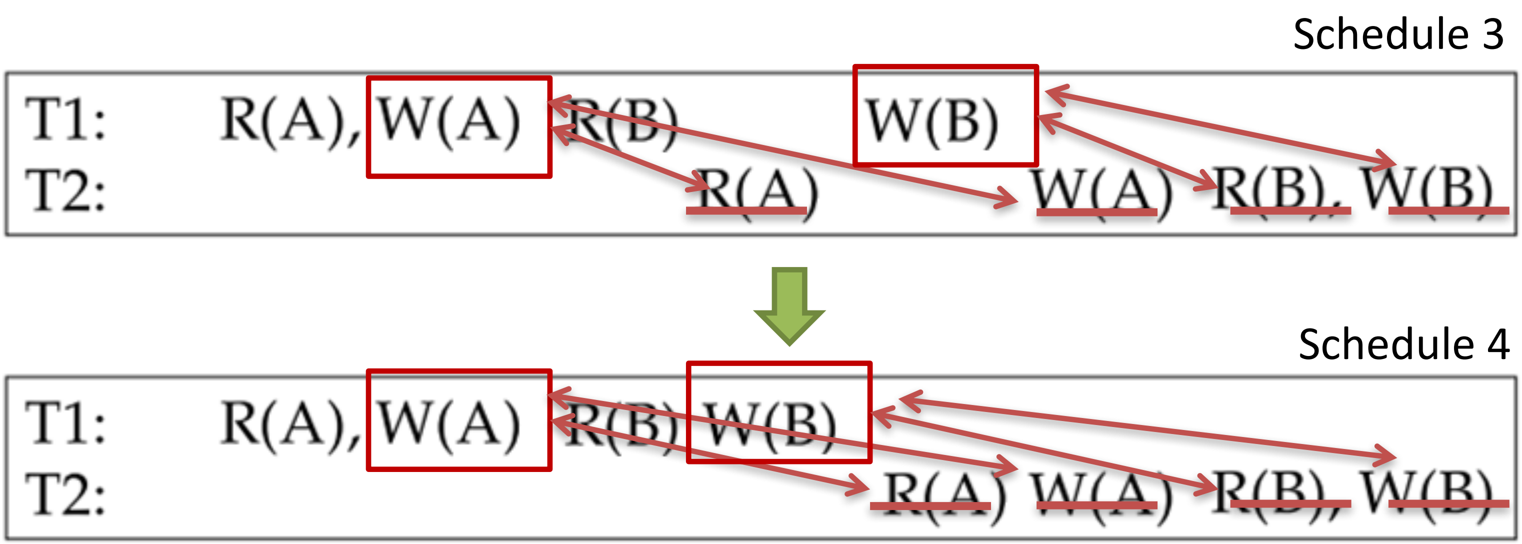
Conflict Equivalent

- If two operations are not conflict, we can ***swap*** them to generate an equivalent schedule
- Schedule 1 is ***conflict equivalent*** to schedule 2 and schedule 3



Conflict Serializable

- By swapping non-conflict operations, we can transfer the schedule 1 into a serial schedule 4
- We say that schedule 1 is ***conflict serializable***



Ensuring Conflict Serializability is ***Not Enough***

T1:	R(A), $\boxed{\text{W(A),}}$		R(B), W(B), Abort
T2:		R(A), W(A), C	

- Conflict serializable, but ***not*** recoverable

Avoiding Anomalies

- We also need to ensure recoverable schedule
- Definition: A schedule is ***recoverable*** if each tx *T* commits only after all txs whose changes *T* reads, commit
- How?
  - Avoid cascading aborts
  - Disallow a tx from reading uncommitted changes from other txs

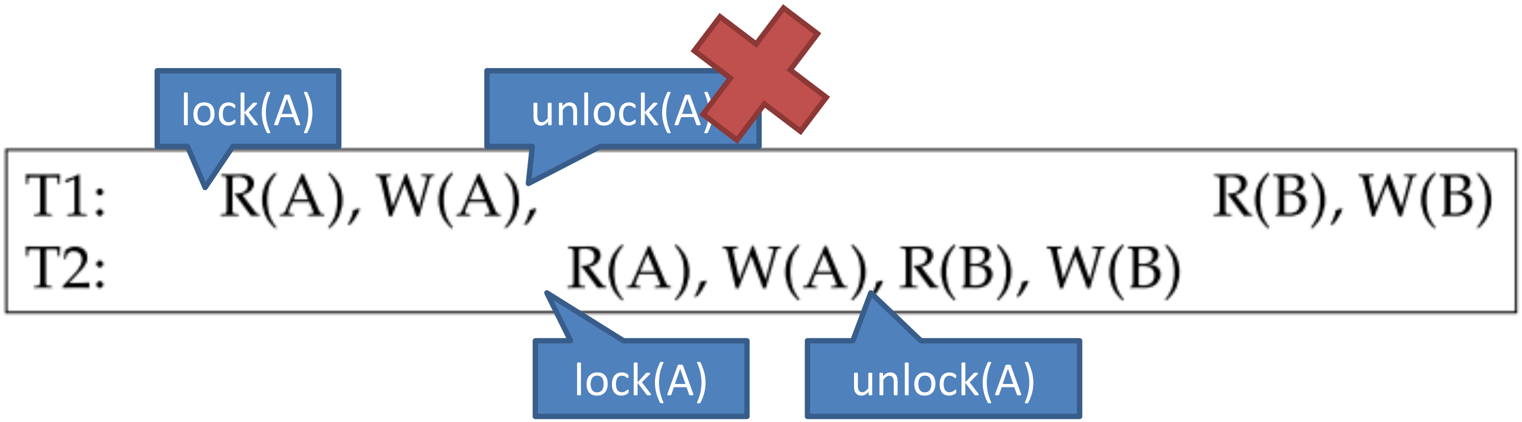
Outline

- Schedules
- Anomalies
- Lock-based concurrency control
  - 2 Phase Locking (2PL)
  - Strict 2 Phase Locking (S2PL)
  - Deadlock

Lock-Based Concurrency Control

- For isolation and consistency, a DBMS should only allow ***serializable***, ***recoverable*** schedules
  – Uncommitted changes cannot be seen (no WR)
  – Ensure repeatable read (no RW)
  – Cannot overwrite uncommitted change (no WW)

- A ***lock*** for each data item seems to be a good solution



Lock $\neq$ latch

- Lock: long-term, tx-level
- Latch: short-term, ds/alg-level

Questions

- What type of lock to get for each operation?
- When should a transaction acquire/release lock?

- We need a ***locking protocol***
  - A set of rules followed by all transactions for requesting and releasing locks

Two phase Locking Protocol (2PL)

- Defines two type of locks:
  - ***Shared (S) lock***
  - ***Exclusive (X) lock***

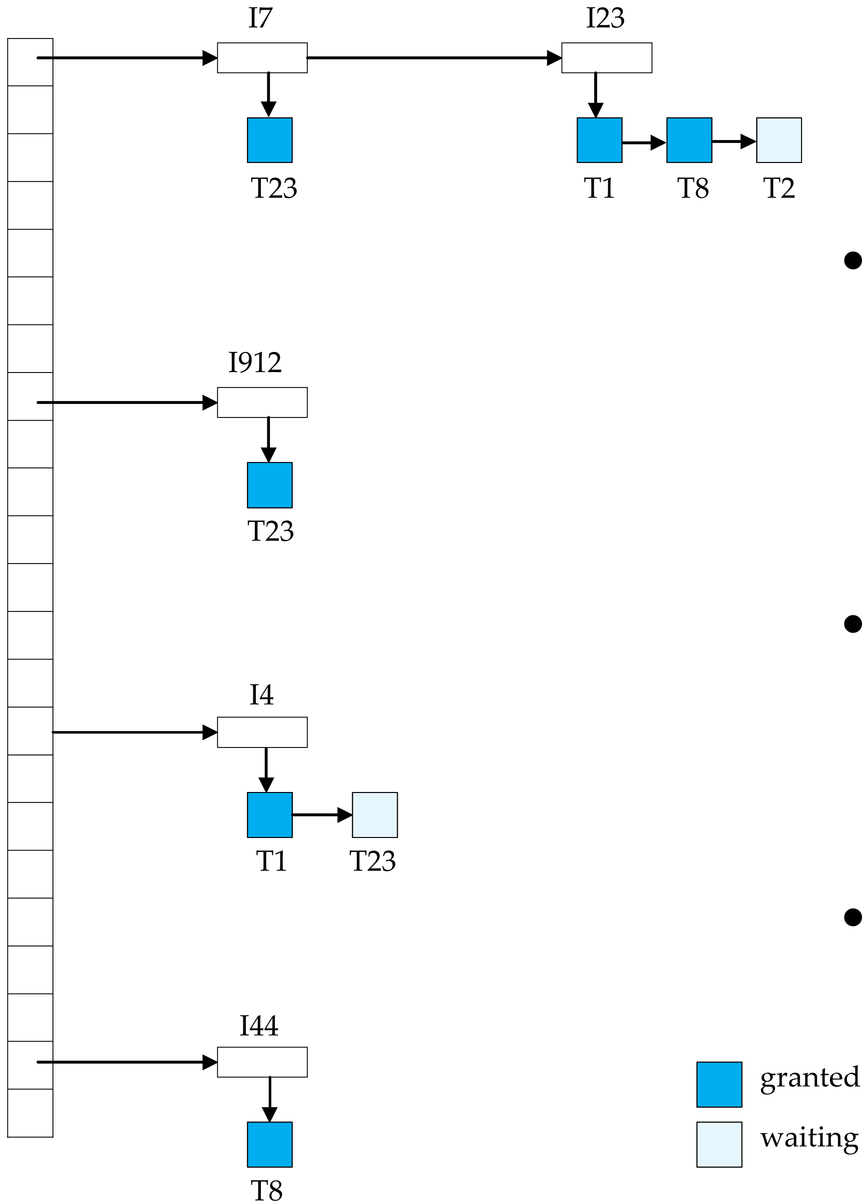
Compatible?	S	X
S	True	False
X	False	False

- Phase 1: Growing Phase
  - Each tx must obtain an S (X) lock on an object before reading (writing) it
- Phase 2: Shrinking Phase
  - A transaction can not request additional locks once it releases any locks
- Ensures conflict serializability

Implementation

- Lock and unlock requests are handled by the ***lock manager***
  - Shared between concurrency managers
- Lock table entry
  - Number of transactions currently holding a lock
  - Type of lock held
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations

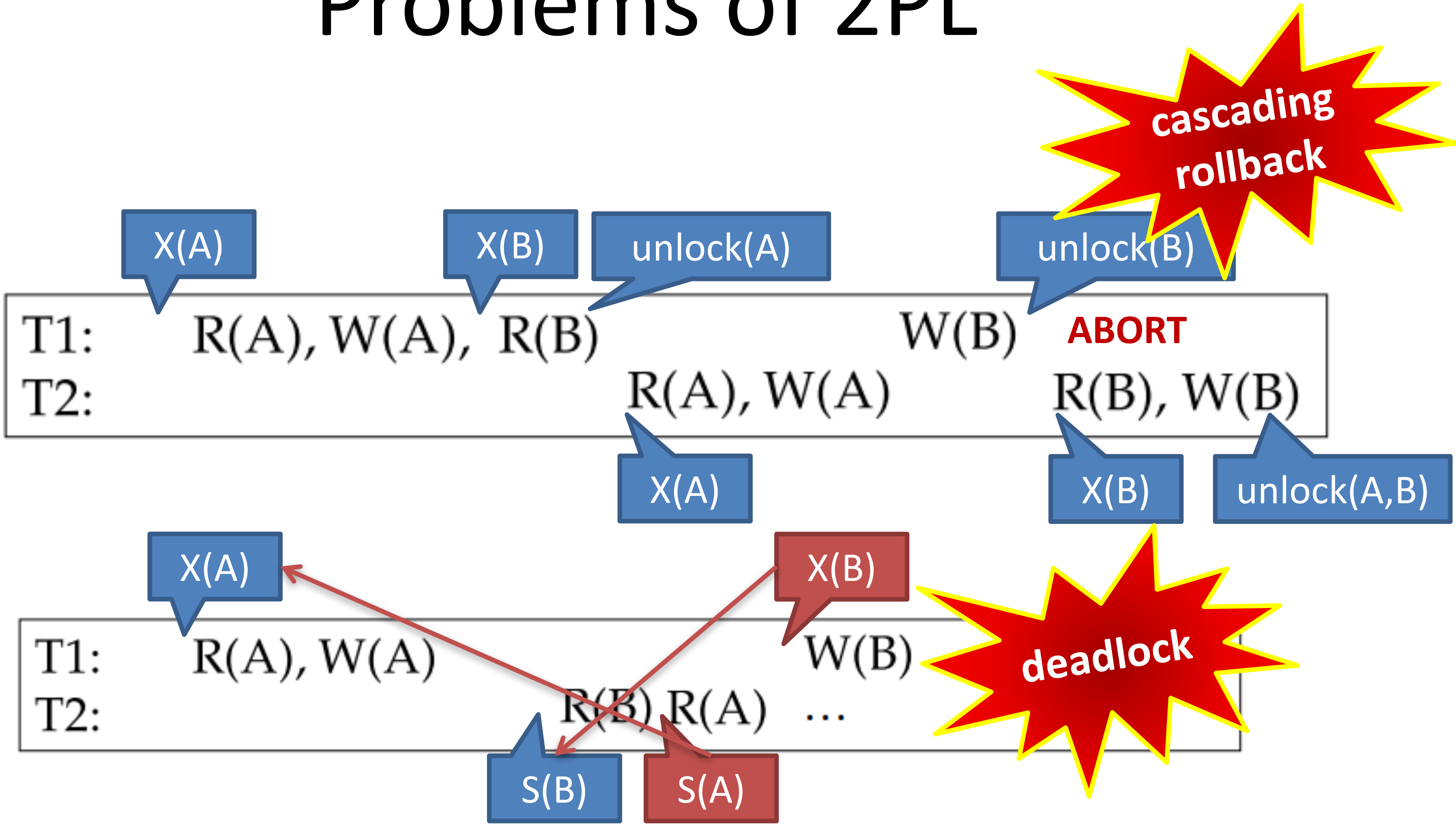
Lock Table



- Implemented as an in-memory hash table indexed on the name of the data item being locked
- New lock request is added to the end of the queue of requests for the data item
- Request is granted if it is compatible with all earlier requests

From Database System Concepts 6/e, Silberschatz, Korth. Sudarshan.

Problems of 2PL



- ***Starvation*** is also possible if concurrency control manager is badly implemented

Outline

- Schedules
- Anomalies
- Lock-based concurrency control
  - 2 Phase Locking (2PL)
  - Strict 2 Phase Locking (S2PL)
  - Deadlock

How to improve 2PL to avoid cascading rollback?

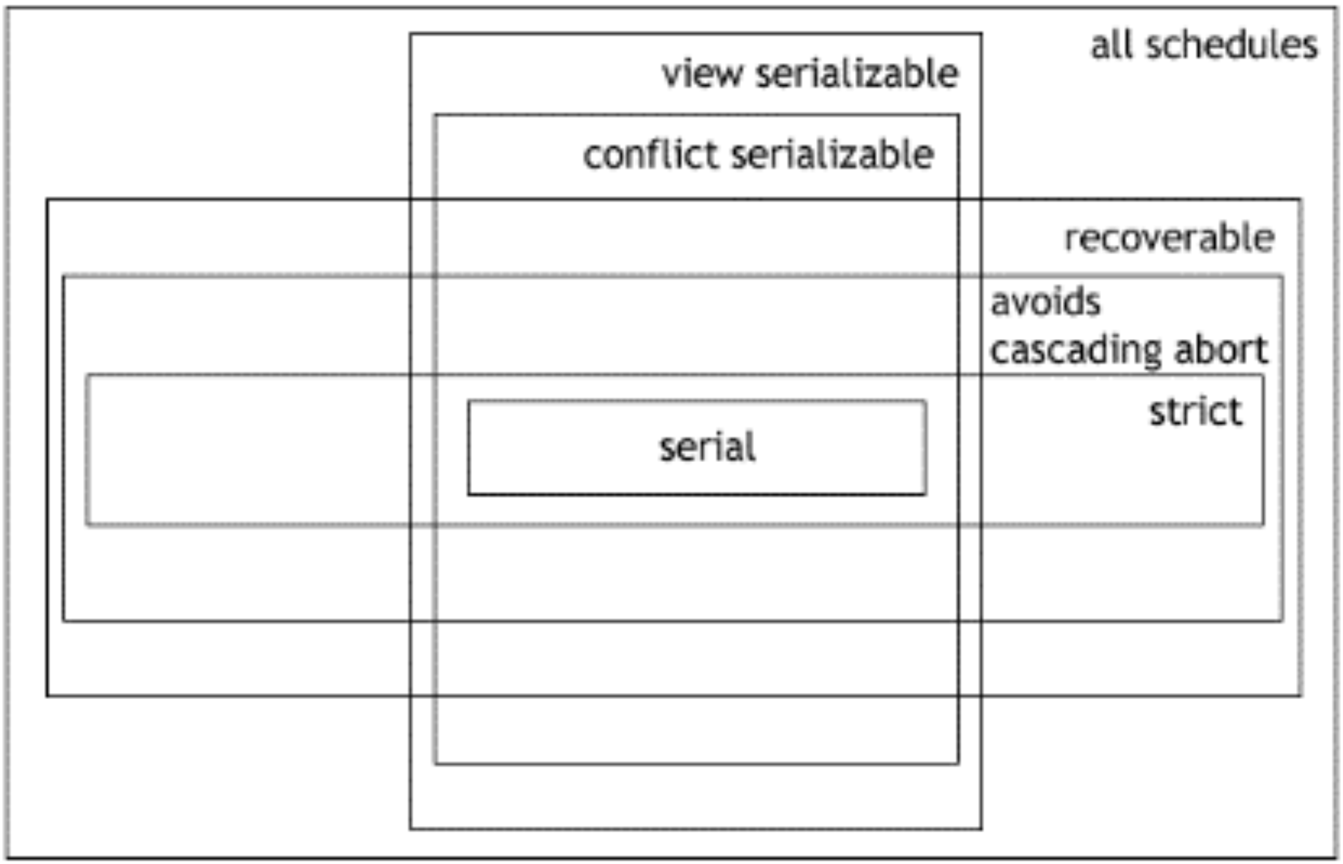
Strict Two-Phase Locking

- S2PL
    1. Each tx obtains locks as in the growing phase in 2PL
    2. But the tx ***holds all locks until it completes***
- Allows only serializable and ***stric*** schedules

Strict Two-Phase Locking

- Definition: A schedule is ***strict*** iff for any two txs T1 and T2, if a write operation of T1 precedes a conflicting operation of T2 (either read or write), then T1 commits before that conflicting operation of T2
  - Strictness → no cascading abort (converse not true)
- Avoids cascading rollback, but still has deadlock

Serializability and Recoverability

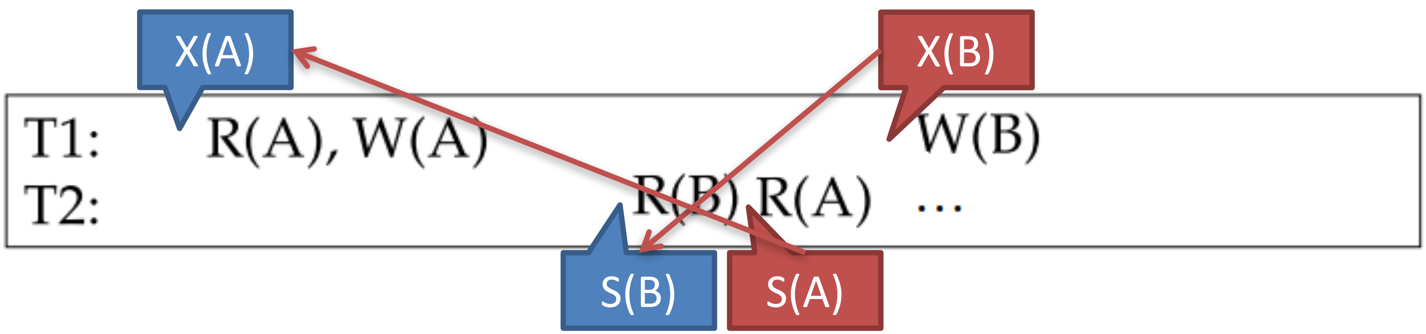


Outline

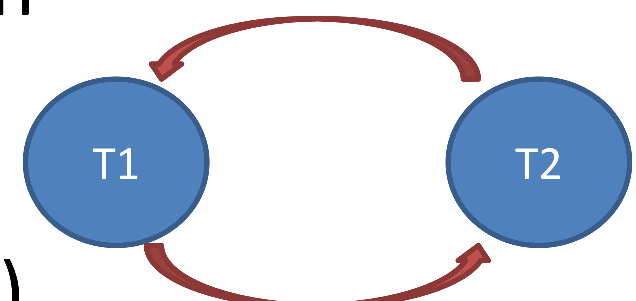
- Schedules
- Anomalies
- Lock-based concurrency control
  - 2 Phase Locking (2PL)
  - Strict 2 Phase Locking (S2PL)
  - Deadlock

Coping with Deadlocks

- Cycle of transactions waiting for locks to be released by each other



- Detection: ***Waits-for*** graph
  - For detecting cycles
- Checked when acquiring locks (or buffers)



Other Techniques (1)

- ***Timeout & rollack*** (deadlock detection)
  - Assume T_i wants a lock that T_j holds
  1. T_i waits for the lock
  2. If T_i stays on the wait list too long then: T_i is rolled back
- ***Wait-die*** (deadlock prevention)
  - Assume each T_i has a timestamp (e.g., tx number)
  - If T_i wants a lock that T_j holds
  1. If T_i is older than T_i, it waits for T_j;
  2. Otherwise T_i aborts

Other Techniques (2)

- ***Conservative locking*** (deadlock prevention)
  - Every T_i locks ***all objects at once*** (atomically) in the beginning
  - No interleaving for conflicting txs---performs well only if there is no/very few long txs (e.g., in-memory DBMS)
  - How to know which objects to lock before tx execution?
  - Requires the coder of a stored procedure to specify its read- and write-sets explicitly
  - Does not support ad-hoc queries

References

- Database Design and Implementation, chapter 14. Edward Sciore.
- Database management System 3/e, chapter 16. Ramakrishnan Gehrke.
- Database system concepts 6/e, chapter 15, 16. Silberschatz.
- Derby Developer's Guide: Locking, concurrency, and isolation.
  - http://db.apache.org/derby/docs/10.9/devguide/cdevconcepts30291.html
- IBM DB2 document: Locks and concurrency control
  - http://publib.boulder.ibm.com/infocenter/db2luw/v9r5/index.jsp?topic=%2Fcom.ibm.db2.luw.admin.perf.doc%2Fdoc%2Fc0005266.html