CS 4604: Introduction to Database Management Systems

Transactions 2: 2PL and Deadlocks

Virginia Tech CS 4604 Sprint 2021

Instructor: Yinlin Chen

Today's Topics

- 2PL/2PLC
- Lock Management
- Deadlocks
  - Detection
  - Prevention
- Specialized Locking

Review

- DBMSs support ACID Transaction semantics
- Concurrency control and Crash Recovery are key components
- For Isolation property, serial execution of transactions is safe but slow
  - Try to find schedules equivalent to serial execution
- One solution for “conflict serializable” schedules is Two Phase Locking (2PL)

Lost update problem - no locks

T1	T2
Read(N)	
	Read(N)
N = N -1	
	N = N -1
Write(N)	
	Write(N)

How Do We Lock Data?

- Not by any crypto or hardware enforcement
  - There are no adversaries here … this is all within the DBMS
- We lock by simple convention:
  - Within DBMS internals, we observe a lock *protocol*
  - If your transaction *holds* a lock, and my transaction *requests* a conflicting lock, then I am queued up waiting for that lock.

Lock

- Simple convention within the DBMS:
  - Each *data element* has a unique lock
  - Each transaction must first acquire the lock before reading/writing that element
  - If the lock is taken by another transaction, then wait
  - The transaction must release the lock(s) at some point
- Different *lock protocols / schemes* differ by:
  - When to lock / unlock each data element
  - What data element to lock
  - What happens when a txn waits for a lock

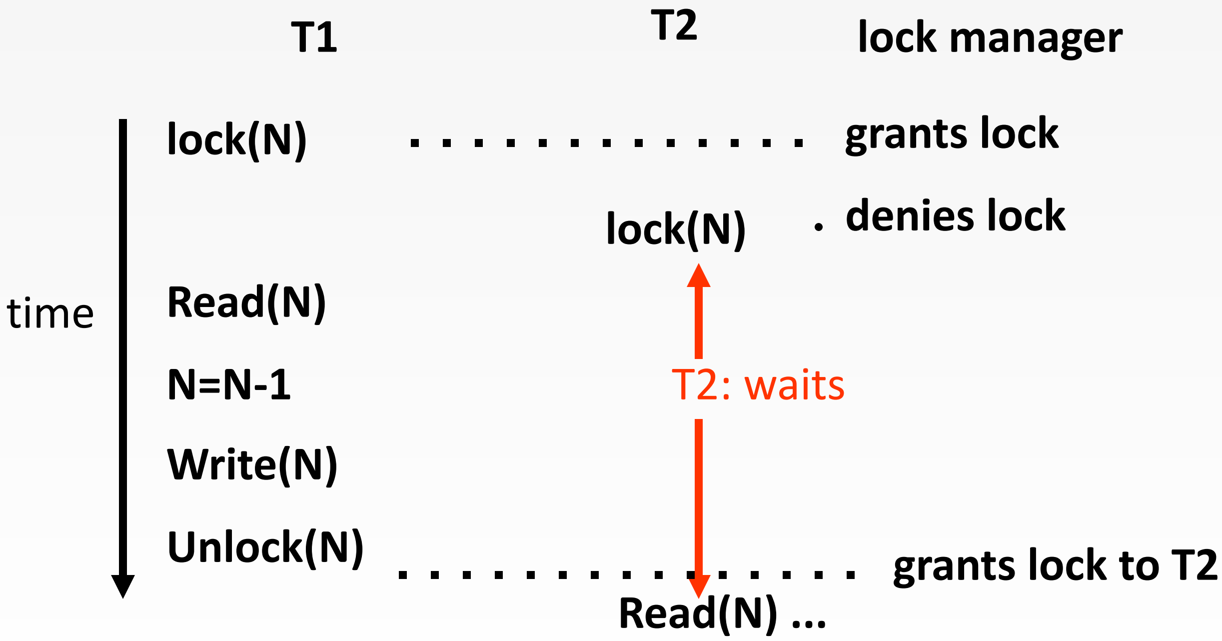
What are "data elements"?

- Major differences between vendors:
  - Lock on the entire database
    - SQLite
  - Lock on individual records
    - SQL Server, DB2, etc
- Lock Granularity
  - Fine granularity locking (e.g., tuples)
    - High concurrency
    - High overhead in managing locks
  - Coarse grain locking (e.g., tables, entire database)
    - Many false conflicts
    - Less overhead in managing locks

Solution – part 1

- with locks:
- lock manager: grants/denies lock requests

Lost update problem – with locks



VIRGINIA TECH

Lock Modes

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)
- Cannot get new locks after releasing any locks (strict 2PL)

Compatibility matrix

	None	S	X
None	✔	✔	✔
S	✔	✔	✖
X	✔	✖	✖

Lock Management

- Lock and unlock requests handled by Lock Manager (LM)
- LM maintains a hashtable, keyed on names of objects being locked.
- LM keeps an entry for each currently held lock
- Entry contains
  - Granted set: Set of xacts currently granted access to the lock
  - Lock mode: Type of lock held (Shared or eXclusive)
  - Wait Queue: Queue of lock requests

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

Lock Management (continued)

- **When lock request arrives:**
  - Does any xact in Granted Set or Wait Queue want a conflicting lock?
    - If no, put the requester into “granted set” and let them proceed
    - If yes, put requester into wait queue (typically **FIFO**)
- **Lock upgrade:**
  - Xact with shared lock can request to upgrade to exclusive

	Granted Set	Mode	Wait Queue
A	{T1, T2}	S	T3(X) ← T4(X)
B	{T6}	X	T5(X) ← T7(S)

Summary: Lock Management

- transactions request locks (or upgrades)
- lock manager grants or blocks requests
- transactions release locks
- lock manager updates lock-table

Locks

- Q: I just need to read ‘N’ - should I still get a lock?

Actions on Locks

- $Lock_i(A)$ / $L_i(A)$ = transaction T_i acquires lock for element A
- $Unlock_i(A)$ / $U_i(A)$ = transaction T_i releases lock for element A

A Non-Serializable Schedule

T1	T2
READ(A)	
A := A+100	
WRITE(A)	
	READ(A)
	A := A*2
	WRITE(A)
	READ(B)
	B := B*2
	WRITE(B)
READ(B)	
B := B+100	
WRITE(B)	

Example

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Using locks has ensured a conflict-serializable schedule

Another Example

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;
$L_1(B)$; READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	

Locks did not **enforce** conflict-serializability

Two Phase Locking (2PL)

- **The most common scheme for enforcing conflict serializability**
- **A bit “pessimistic”**
  - Sets locks for fear of conflict… Some cost here.
  - Alternative schemes use multiple versions of data and “optimistically” let transactions move forward
    - Abort when conflicts are detected.
    - Some names to know/look up:
      - Optimistic Concurrency Control
      - Timestamp-Ordered Multiversion Concurrency Control
    - We will not study these schemes in this lecture

Two Phase Locking (2PL), Part 2

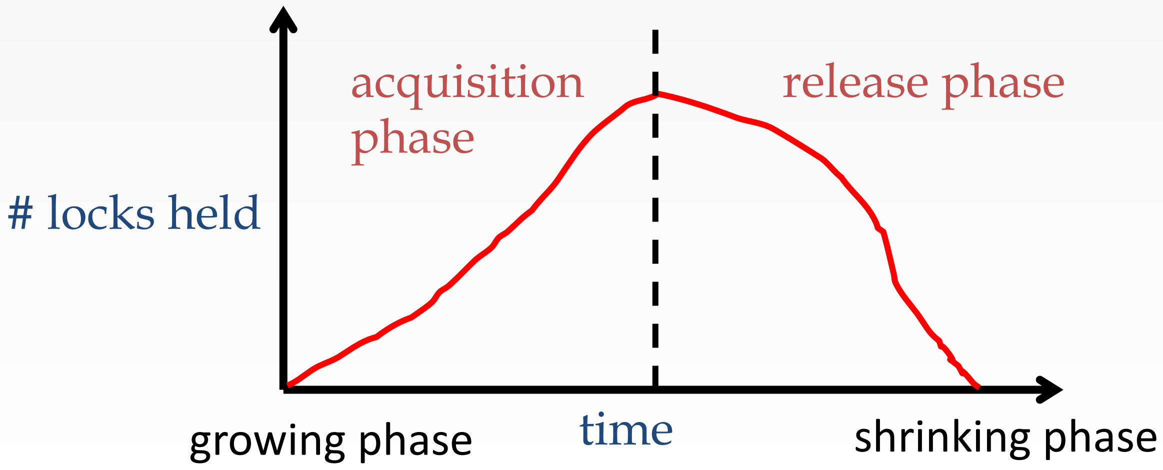
- Rules:
  - Xact must obtain a S (shared) lock before reading, and an X (exclusive) lock before writing.
  - **Xact cannot get new locks after releasing any locks**

Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

Two Phase Locking (2PL), Part 3

- **2PL guarantees conflict serializability**
- But, does not prevent **cascading aborts**

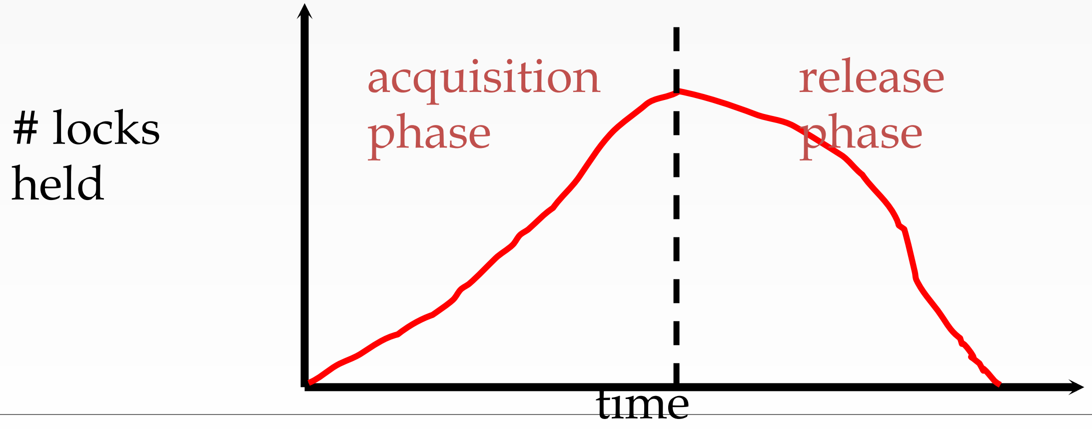


Why 2PL guarantees conflict serializability

- When a committing transaction has reached the end of its acquisition phase…
  - Call this the “lock point”
  - At this point, it has *everything it needs* locked…
  - … and any conflicting transactions either:
    - started release phase before this point
    - are blocked waiting for this transaction
- Visibility of actions of two conflicting transactions are ordered by their lock points
- The order of lock points gives us an equivalent serial schedule!

Two-Phase Locking (2PL), cont.

- 2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic), but, it is subject to Cascading Aborts.



Strict Two Phase Locking (2PL)

- **Problem: Cascading Aborts**
- Example: rollback of T1 requires rollback of T2!

T1:	**R(A), W(A)**		**Abort**
T2:		**R(A), W(A)**	

- Solution: Strict 2PL, i.e, keep all locks, until 'commit'

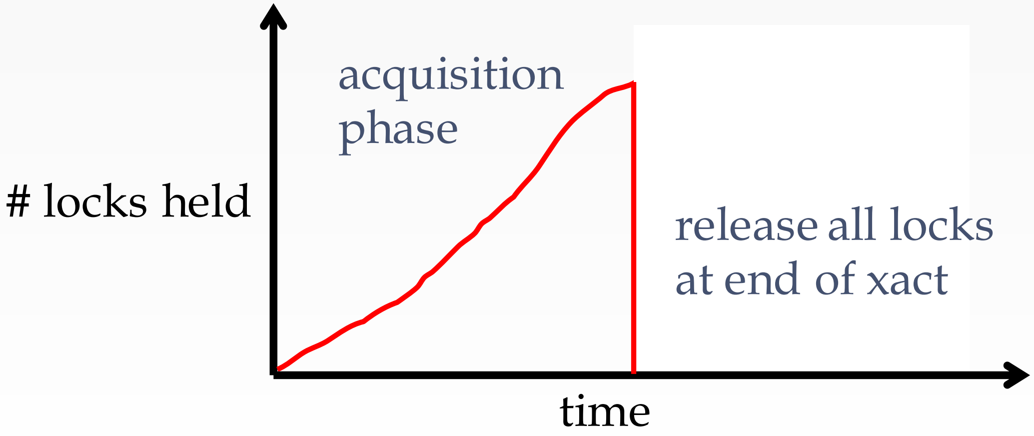
Non-recoverable Schedule

T1	T2
L_1(A); L_1(B); READ(A)	
A :=A+100	
WRITE(A); U_1(A)	
	L_2(A); READ(A)
	A := A*2
	WRITE(A);
	L_2(B); BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); U_1(B);	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); U_2(A); U_2(B);
	Commit
Rollback	

Aka cascading aborts

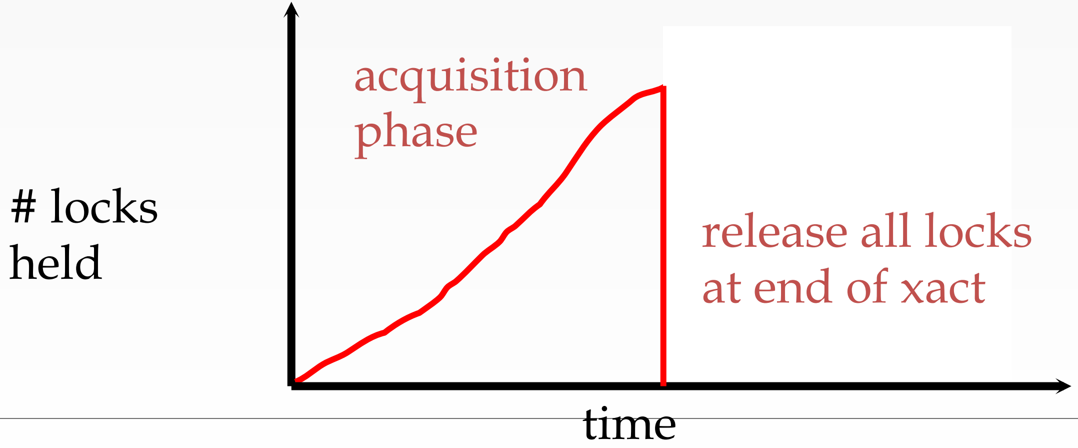
Strict Two Phase Locking

- Same as 2PL, except all locks released together when transaction completes
  - (i.e.) either
    - Transaction has committed (all writes durable), OR
    - Transaction has aborted (all writes have been undone)



Strict 2PL == 2PLC (2PL till Commit)

- In effect, "shrinking phase" is delayed until
  - Transaction commits (commit log record on disk), or
  - Aborts (then locks can be released after rollback).



VIRGINIA TECH

Strict 2PL

T1	T2
$L_1(A)$; READ(A)	
A :=A+100	
WRITE(A);	
	$L_2(A)$; BLOCKED…
$L_1(B)$; READ(B)	
B :=B+100	
WRITE(B);	
Rollback & $U_1(A)$;$U_1(B)$;	
	…GRANTED; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B);
	Commit & $U_2(A)$; $U_2(B)$;

Strict 2PL

- Lock-based systems always use strict 2PL
- Easy to implement:
  - Before a transaction reads or writes an element A, insert an L(A)
  - When the transaction commits/aborts, then release all locks
- Ensures both conflict serializability and recoverability

Non-2PL, A = 1000, B = 2000, Output = ?

T1	T2
Lock_X(A)	
Read(A)	
	Lock_S(A)
A: = A-50	
Write(A)	
Unlock(A)	
	Read(A)
	Unlock(A)
	Lock_S(B)
Lock_X(B)	
	Read(B)
	Unlock(B)
	PRINT(A), PRINT(B), PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

Output: 950, 2000, 2950

VIRGINIA TECH

Non-2PL, A = 1000, B = 2000, Output = ? cont

T1	T2
Lock_X(A)	
Read(A): (A=1000)	
	Lock_S(A)
A: = A-50 (A=950)	
Write(A) A=950	
Unlock(A)	
	Read(A) (A = 950)
	Unlock(A)
	Lock_S(B)
Lock_X(B)	
	Read(B) (B=2000)
	Unlock(B)
	PRINT(A), PRINT(B), PRINT(A+B)
Read(B) (B=2000)	
B := B +50 (B=2050)	
Write(B) B=2050	
Unlock(B)	

Output: 950, 2000, 2950

2PL, A = 1000, B = 2000, Output = ?

T1	T2
Lock_X(A)	
Read(A)	
A: = A-50	
Write(A)	
Unlock(A)	
Lock_X(B)	
	Lock_S(A)
	Read(A)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	
	Unlock(A)
	Lock_S(B)
	Read(B)
	Unlock(B)
	PRINT(A), PRINT(B), PRINT(A+B)

Output: 950, 2050, 3000

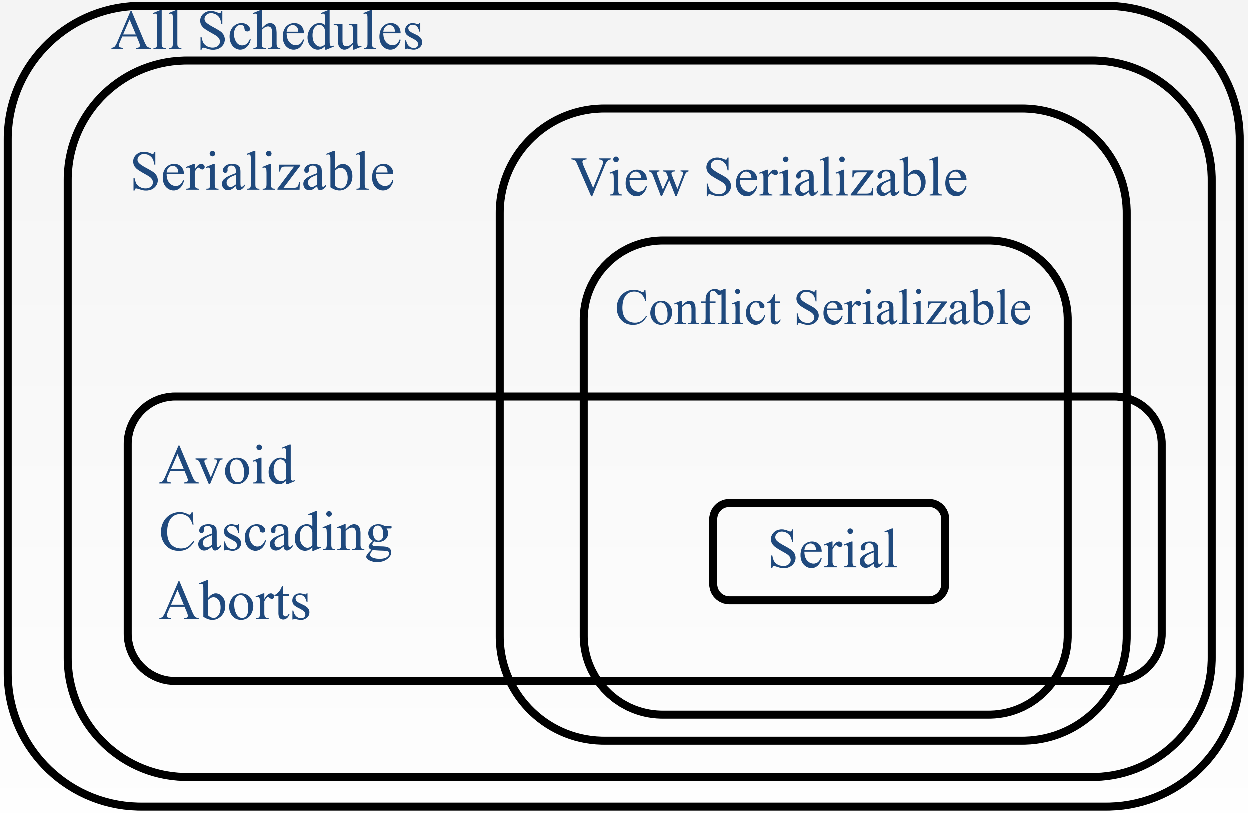
Strict 2PL, A = 1000, B = 2000, Output = ?

T1	T2
Lock_X(A)	
Read(A)	
	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A), PRINT(B), PRINT(A+B)
	Unlock(A)
	Unlock(B)

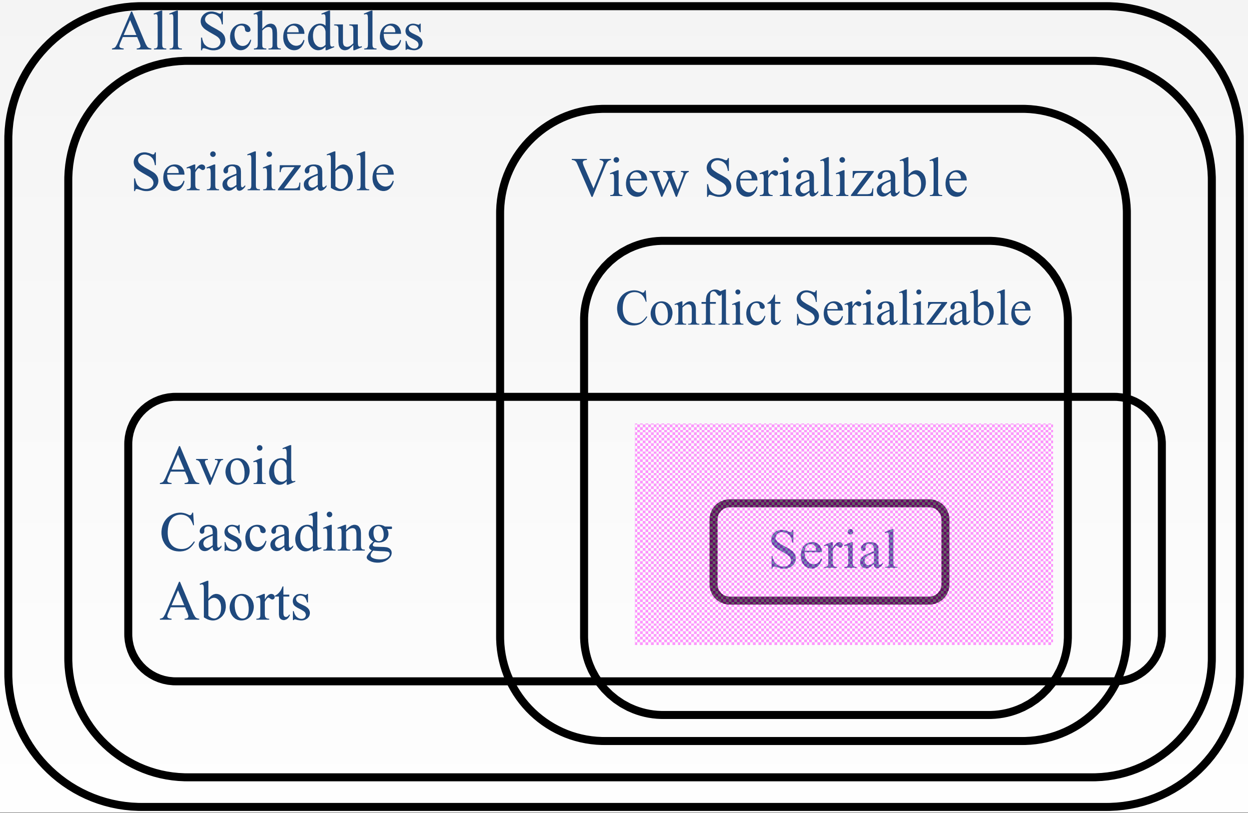
Output: 950, 2050, 3000

VIRGINIA TECH™

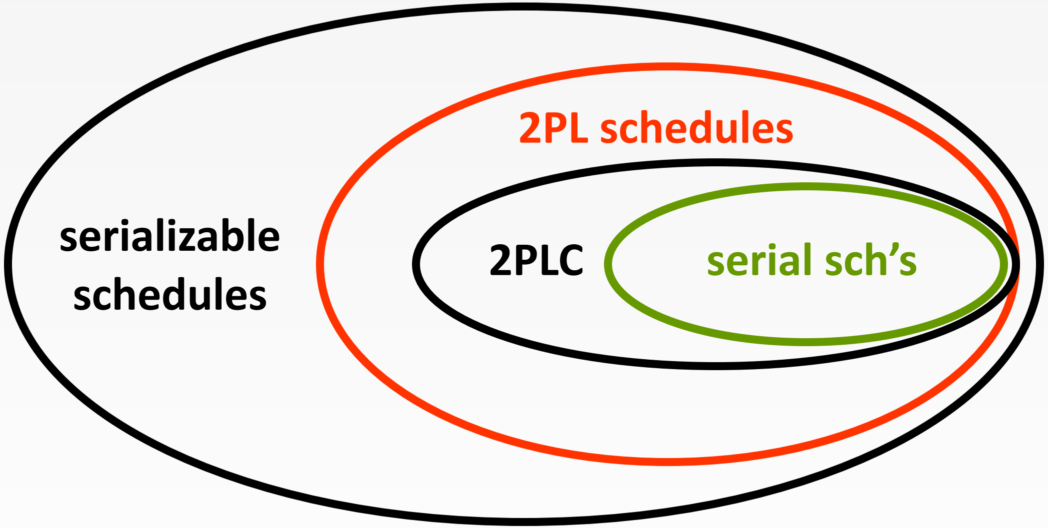
Venn Diagram for Schedules



Q: Which schedules does Strict 2PL allow?

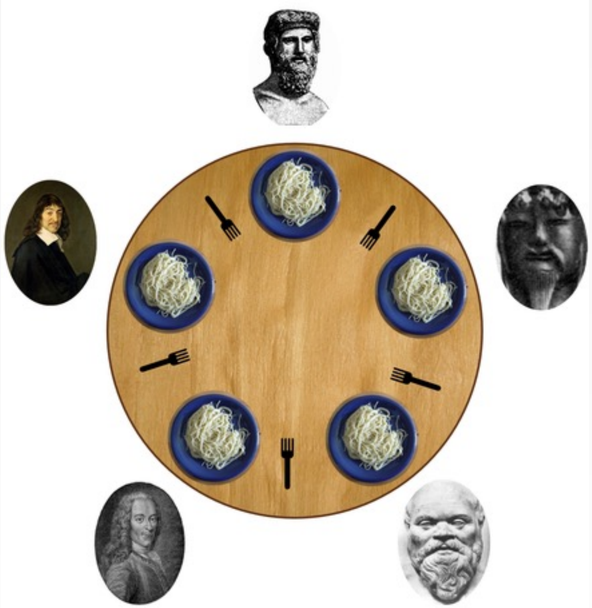


Another Venn diagram



Another problem: Deadlocks

- T1: R(A), W(B)
- T2: R(B), W(A)
- T1 holds the lock on A, waits for B
- T2 holds the lock on B, waits for A



Deadlocks

- Deadlock: Cycle of Xacts waiting for locks to be released by each other.
- Three ways of dealing with deadlocks:
  - Prevention
  - Avoidance
  - Detection and Resolution
- Many systems just punt and use timeouts
  - What are the dangers with this approach?

Deadlock Detection

- Create and maintain a **waits-for** graph:
  - Nodes are transactions
  - Edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in waits-for graph

Deadlock Detection, Part 2

Example:

T1:
T2:
T3:

T4:

Deadlock Detection, Part 3

Example:

T1: S(A)
T2:
T3:
T4:

Deadlock Detection, Part 4

Example:

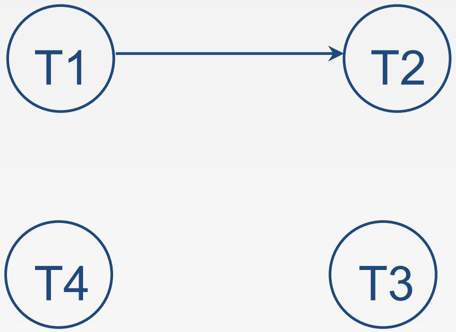
T1: S(A) S(D)
T2:
T3:
T4:

Deadlock Detection, Part 5

Example:

T1:	S(A)	S(D)	
T2:			X(B)
T3:			
T4:			

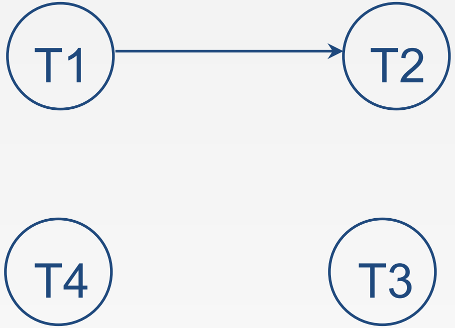
Deadlock Detection, Part 6



Example:

T1:	S(A)	S(D)		S(B)
T2:			X(B)	
T3:				
T4:				

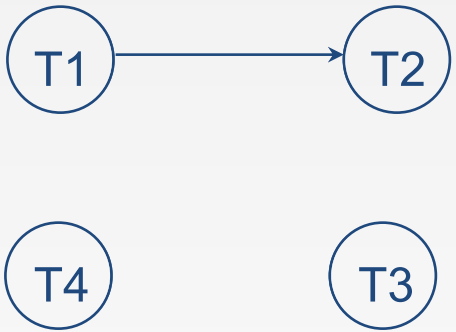
Deadlock Detection, Part 7



Example:

T1:	S(A)	S(D)		S(B)	
T2:			X(B)		
T3:					S(D)
T4:					

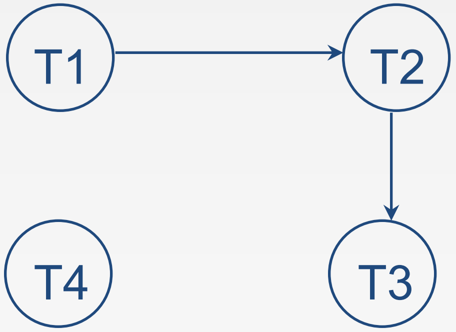
Deadlock Detection, Part 8



Example:

T1: S(A) S(D) S(B)
T2: X(B)
T3: S(D), S(C)
T4:

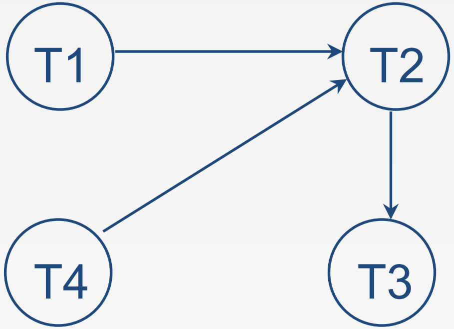
Deadlock Detection, Part 9



Example:

T1:	S(A)	S(D)		S(B)			
T2:			X(B)				X(C)
T3:					S(D)	S(C)	
T4:							

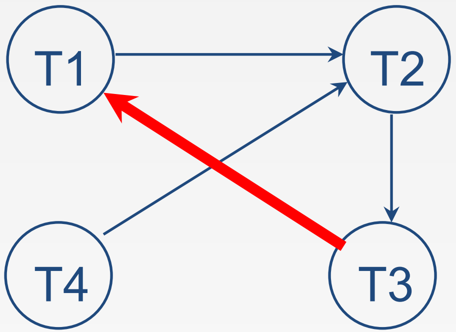
Deadlock Detection, Part 10



Example:

T1:	S(A)	S(D)		S(B)				
T2:			X(B)				X(C)	
T3:					S(D)	S(C)		
T4:								X(B)

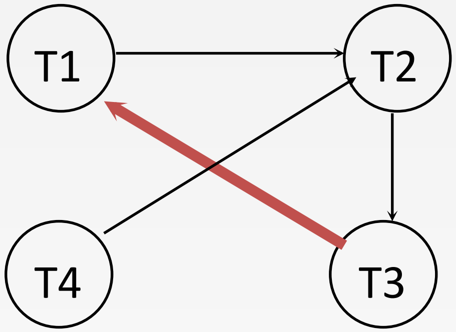
Deadlock Detection, Part 11



Example:

T1: S(A) S(D) S(B)
T2: X(B) X(C)
T3: S(D) S(C) X(A)
T4: X(B)

Deadlock Detection, Part 12



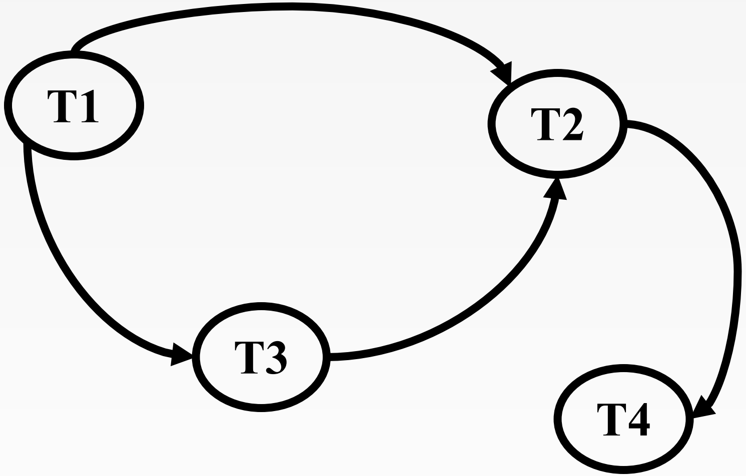
Example:

T1: S(A), S(D), S(B)
T2: X(B) X(C)
T3: S(D), S(C), X(A)
T4: X(B)

Deadlock!

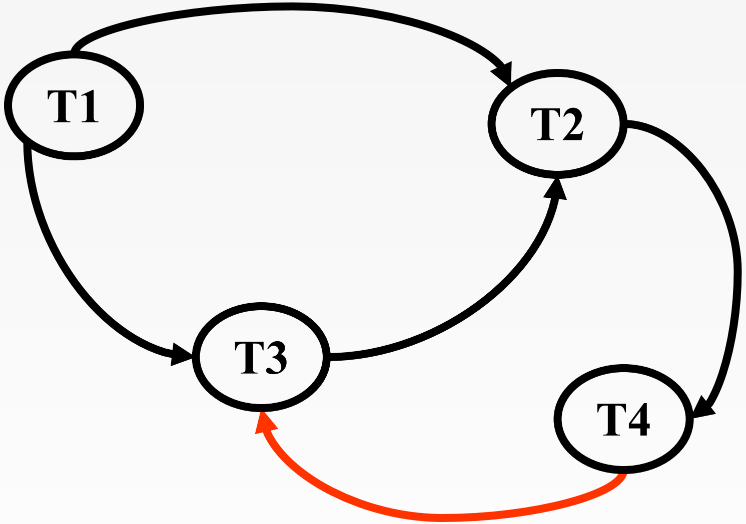
- T1, T2, T3 are deadlocked
  - Doing no good, and holding locks
- T4 still cruising
- In the background, run a deadlock detection algorithm
  - Periodically extract the waits-for graph
  - Find **cycles**
  - “Shoot” a transaction on the cycle
- Empirical fact
  - Most deadlock cycles are small (2-3 transactions)

Another example



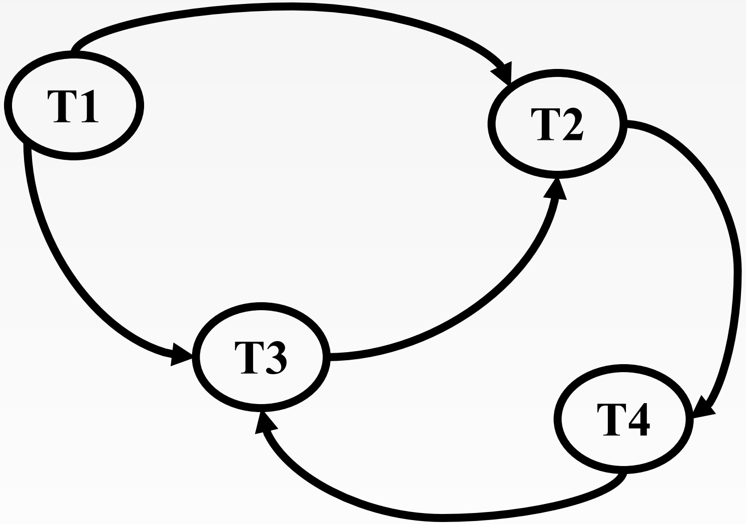
- is there a deadlock?
- if yes, which acts are involved?

Another example



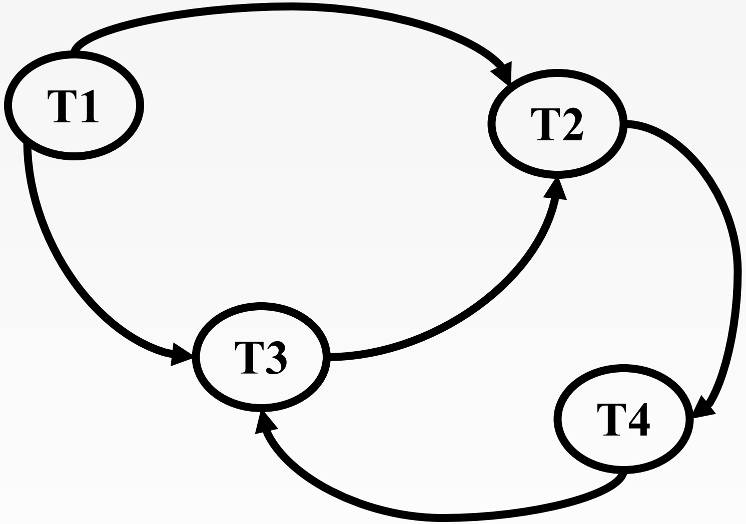
- now, is there a deadlock?
- if yes, which xacts are involved?

Deadlock handling



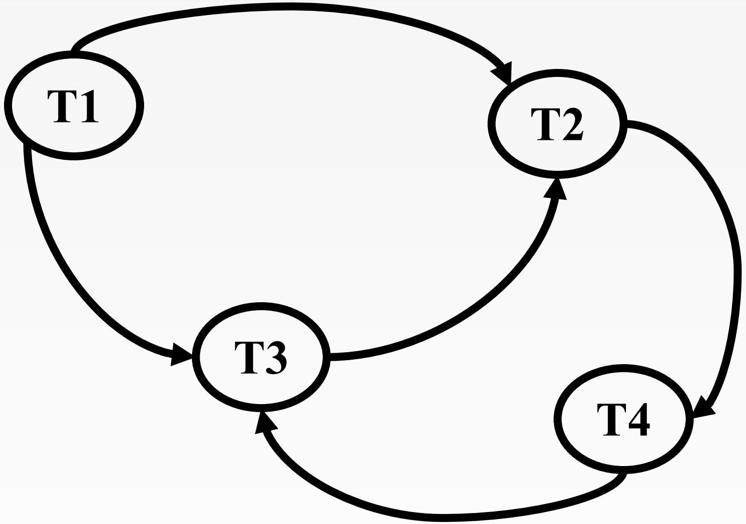
- Q: what to do?

Deadlock handling



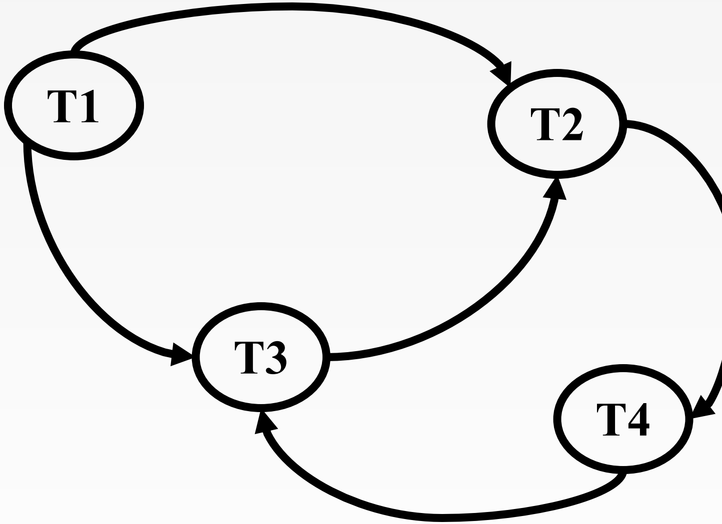
- Q0: what to do?
  - A: select a 'victim' & 'rollback'
- Q1: which/how to choose?

Deadlock handling



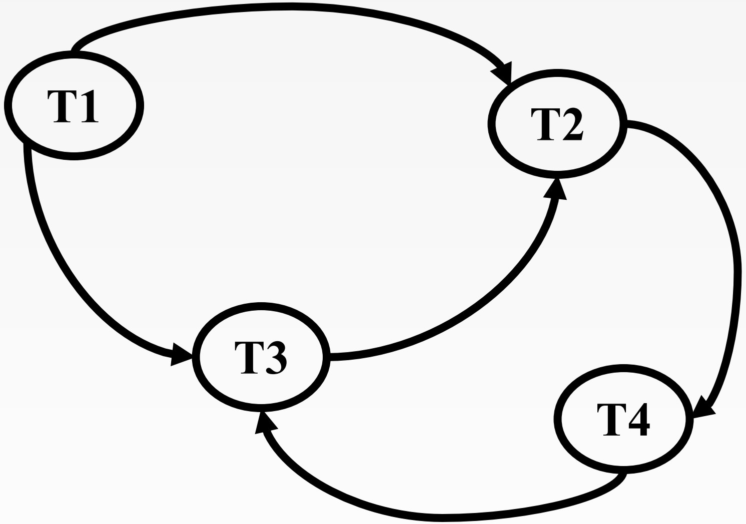
- Q1: which/how to choose?
  - A1.1: by age
  - A1.2: by progress
  - A1.3: by # items locked already...
  - A1.4: by # xacts to rollback
- Q2: How far to rollback?

Deadlock handling



- Q2: How far to rollback?
  - A2.1: completely
  - A2.2: minimally
- Q3: Starvation??

Deadlock handling

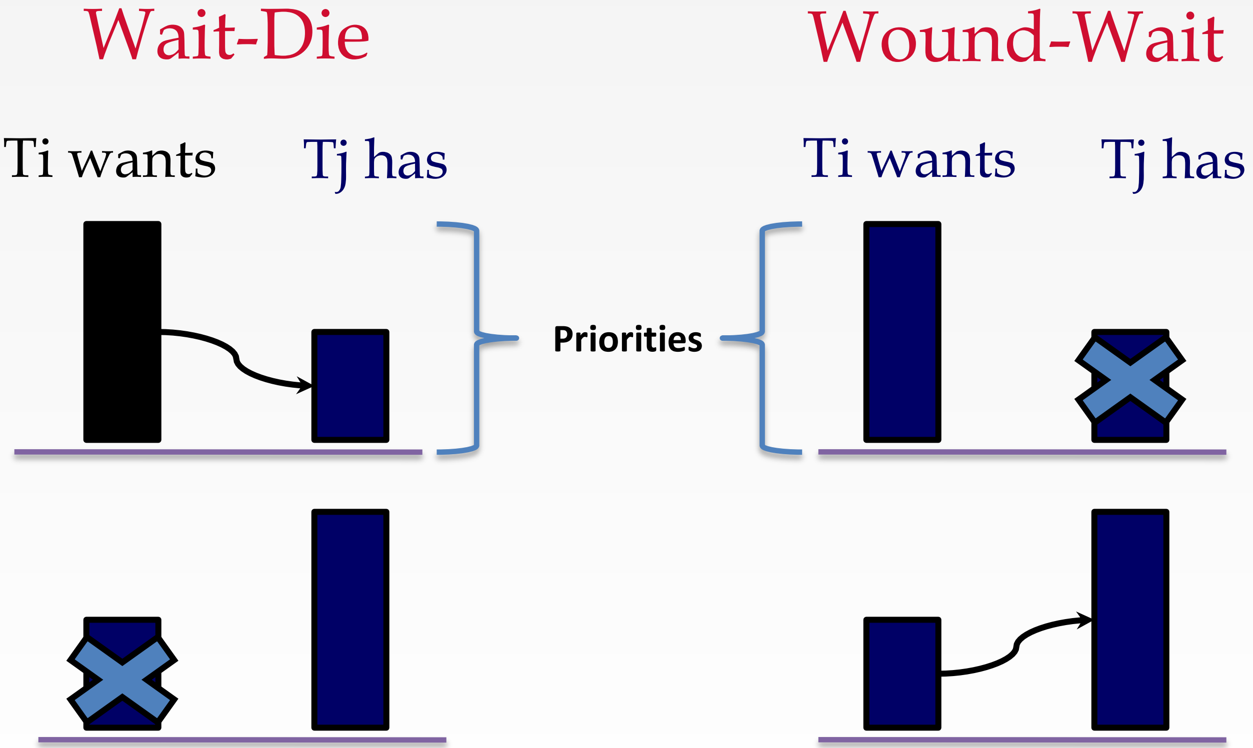


- Q3: Starvation??
- A3.1: include #rollbacks in victim selection criterion.

Deadlock Prevention

- Assign priorities based on age (timestamps)
  - older -> higher priority
- We only allow ‘old-wait-for-young’
- (or only allow ‘young-wait-for-old’)
- and rollback violators. Specifically:
- Say Ti wants a lock that Tj holds - two policies:
  - Wait-Die: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts (ie., old wait for young)
  - Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits (ie., young wait for old)

Deadlock Prevention



Deadlock Prevention

- Q: Why do these schemes guarantee no deadlocks?
- A: only one ‘type’ of direction allowed.
- Q: When a transaction restarts, what is its (new) priority?
- A: its original timestamp. -- Why?

SQL statement

- usually, conc. control is transparent to the user, but
- LOCK <table-name> [EXCLUSIVE|SHARED]

Phantom Problem

- So far we have assumed the database to be a *static* collection of elements (=tuples)
- If tuples are inserted/deleted then the *phantom problem* appears
- A “phantom” is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears !

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

T1 sees a “phantom” product A3

TRANSACTION

START TRANSACTION

START TRANSACTION — start a transaction block

Synopsis

```
START TRANSACTION [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

Transaction Support in SQL-92

recommended

- **SERIALIZABLE** – No phantoms, all reads repeatable, no “dirty” (uncommited) reads.
- REPEATABLE READS – phantoms may happen.
- READ COMMITTED – phantoms and unrepeatable reads may happen
- READ UNCOMMITTED – all of them may happen.

Transaction Support in SQL-92

- SERIALIZABLE : obtains all locks first; plus index locks, plus strict 2PL
- REPEATABLE READS – as above, but no index locks
- READ COMMITTED – as above, but S-locks are released immediately
- READ UNCOMMITTED – as above, but allowing ‘dirty reads’ (no S-locks)

Transaction Support in SQL-92

- SET TRANSACTION ISOLATION LEVEL SERIALIZABLE READ ONLY

- Defaults:
- SERIALIZABLE ← isolation level
- READ WRITE ← access mode

Conclusions

- 2PL/2PL-C (=Strict 2PL): extremely popular
- Deadlock may still happen
  - detection: wait-for graph
  - prevention: abort some xacts, defensively
- philosophically: concurrency control uses:
  - locks
  - and aborts

Reading and Next Class

- Transactions Part 2: 2PL/2PLC and Deadlocks: Ch 17
- Next: Logging and Recovery Part 1: Ch 16, 18