Transactions 2: 2PL and Deadlocks
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

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(N)</td>
<td>Read(N)</td>
</tr>
<tr>
<td>( N = N - 1 )</td>
<td>( N = N - 1 )</td>
</tr>
<tr>
<td>Write(N)</td>
<td>Write(N)</td>
</tr>
</tbody>
</table>
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

T1
- lock(N)
- Read(N)
- N = N - 1
- Write(N)
- Unlock(N)

T2
- lock(N)
- T2: waits
- grants lock to T2
- grants lock

Lock manager
- grants lock
- denies lock
Lock Modes

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

Compatibility matrix

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>S</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>X</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Diagram: [Insert Diagram Image]
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

<table>
<thead>
<tr>
<th>Granted Set</th>
<th>Mode</th>
<th>Wait Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>S</td>
<td>T3(X) ← T4(X)</td>
</tr>
<tr>
<td>B</td>
<td>X</td>
<td>T5(X) ← T7(S)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>Granted Set</th>
<th>Mode</th>
<th>Wait Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>{T1, T2}</td>
<td>S</td>
<td>T3(X) ← T4(X)</td>
</tr>
<tr>
<td>B</td>
<td>{T6}</td>
<td>X</td>
<td>T5(X) ← T7(S)</td>
</tr>
</tbody>
</table>
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

- $\text{Lock}_i(A) / L_i(A) = \text{transaction } T_i$ acquires lock for element $A$
- $\text{Unlock}_i(A) / U_i(A) = \text{transaction } T_i$ releases lock for element $A$
## A Non-Serializable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>READ(A)</strong></td>
<td><strong>READ(A)</strong></td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td><strong>WRITE(A)</strong></td>
<td><strong>WRITE(A)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>READ(B)</strong></td>
</tr>
<tr>
<td></td>
<td>B := B * 2</td>
</tr>
<tr>
<td></td>
<td><strong>WRITE(B)</strong></td>
</tr>
<tr>
<td><strong>READ(B)</strong></td>
<td></td>
</tr>
<tr>
<td>B := B + 100</td>
<td></td>
</tr>
<tr>
<td><strong>WRITE(B)</strong></td>
<td></td>
</tr>
</tbody>
</table>
Using locks has ensured a conflict-serializable schedule.
Another Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1(A)$; READ(A)</td>
<td>$L_2(A)$; READ(A)</td>
</tr>
<tr>
<td>A := A+100</td>
<td>A := A*2</td>
</tr>
<tr>
<td>WRITE(A); $U_1(A)$;</td>
<td>WRITE(A); $U_2(A)$;</td>
</tr>
<tr>
<td></td>
<td>$L_2(B)$; READ(B)</td>
</tr>
<tr>
<td></td>
<td>B := B*2</td>
</tr>
<tr>
<td></td>
<td>WRITE(B); $U_2(B)$;</td>
</tr>
<tr>
<td>$L_1(B)$; READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B+100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B); $U_1(B)$;</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Virginia Tech
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)  
  T2: R(A), W(A)  
  Abort

• Solution: Strict 2PL, i.e, keep all locks, until ‘commit’
# Non-recoverable Schedule

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(L_1(A); L_1(B)); READ(A)</td>
<td>(L_2(A)); READ(A)</td>
</tr>
<tr>
<td>A := A + 100</td>
<td>A := A * 2</td>
</tr>
<tr>
<td>WRITE(A); (U_1(A))</td>
<td>WRITE(A);</td>
</tr>
<tr>
<td>(L_2(B)); BLOKCED…</td>
<td>(L_2(B));</td>
</tr>
<tr>
<td>READ(B)</td>
<td>(\ldots\text{GRANTED};) READ(B)</td>
</tr>
<tr>
<td>B := B + 100</td>
<td>B := B * 2</td>
</tr>
<tr>
<td>WRITE(B); (U_1(B))</td>
<td>WRITE(B); (U_2(A)); (U_2(B))</td>
</tr>
</tbody>
</table>

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).
### Strict 2PL

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L_1(A); ) READ(A)</td>
<td>( L_2(A); ) BLOCKED…</td>
</tr>
<tr>
<td>A := A + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(A);</td>
<td></td>
</tr>
<tr>
<td>( L_1(B); ) READ(B)</td>
<td></td>
</tr>
<tr>
<td>B := B + 100</td>
<td></td>
</tr>
<tr>
<td>WRITE(B);</td>
<td></td>
</tr>
<tr>
<td>Rollback &amp; ( U_1(A) ); ( U_1(B) );</td>
<td>...GRANTED; ( \text{READ}(A) )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( A := A \times 2 )</td>
</tr>
<tr>
<td></td>
<td>WRITE(A);</td>
</tr>
<tr>
<td></td>
<td>( L_2(B); ) READ(B)</td>
</tr>
<tr>
<td></td>
<td>( B := B \times 2 )</td>
</tr>
<tr>
<td></td>
<td>WRITE(B);</td>
</tr>
<tr>
<td></td>
<td>Commit &amp; ( U_2(A) ); ( U_2(B) );</td>
</tr>
</tbody>
</table>
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 = ?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock_X(A)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Read(A)</strong></td>
<td><strong>Lock_S(A)</strong></td>
</tr>
<tr>
<td><strong>A: = A - 50</strong></td>
<td><strong>Lock_S(B)</strong></td>
</tr>
<tr>
<td><strong>Write(A)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Unlock(A)</strong></td>
<td><strong>Unlock(A)</strong></td>
</tr>
<tr>
<td><strong>Lock_X(B)</strong></td>
<td><strong>Lock_S(B)</strong></td>
</tr>
<tr>
<td><strong>Read(B)</strong></td>
<td><strong>Unlock(B)</strong></td>
</tr>
<tr>
<td><strong>Unlock(B)</strong></td>
<td><strong>PRINT(A), PRINT(B), PRINT(A + B)</strong></td>
</tr>
<tr>
<td><strong>Read(B)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>B := B + 50</strong></td>
<td><strong>Output: 950, 2000, 2950</strong></td>
</tr>
<tr>
<td><strong>Write(B)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Unlock(B)</strong></td>
<td></td>
</tr>
</tbody>
</table>
Non-2PL, A = 1000, B = 2000, Output = ? cont

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lock_X(A)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>Read(A):</td>
<td>(A=1000)</td>
<td></td>
</tr>
<tr>
<td>A: = A-50</td>
<td>(A=950)</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td>A=950</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unlock(A)</td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td></td>
<td>Lock_S(B)</td>
</tr>
<tr>
<td>Read(B):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B:= B +50</td>
<td>(B=2050)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Write(B)</td>
<td>B=2050</td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>Unlock(B)</td>
<td></td>
<td>PRINT(A), PRINT(B), PRINT(A+B)</td>
</tr>
</tbody>
</table>

Output: 950, 2000, 2950
2PL, A = 1000, B = 2000, Output = ?

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock_X(A)</td>
<td></td>
</tr>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>A := A - 50</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td></td>
</tr>
<tr>
<td>Unlock(A)</td>
<td></td>
</tr>
<tr>
<td>Lock_X(B)</td>
<td>Lock_S(A)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>B := B + 50</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Unlock(B)</td>
<td>Unlock(A)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock(B)</td>
</tr>
<tr>
<td>Output: 950, 2050, 3000</td>
<td>PRINT(A), PRINT(B), PRINT(A+B)</td>
</tr>
</tbody>
</table>
Strict 2PL, A = 1000, B = 2000, Output = ?

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Lock_X(A)</strong></td>
<td><strong>Lock_S(A)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Read(A)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>A := A - 50</strong></td>
<td><strong>Write(A)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Write(A)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Lock_X(B)</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Read(B)</strong></td>
<td><strong>B := B + 50</strong></td>
</tr>
<tr>
<td></td>
<td><strong>B := B + 50</strong></td>
<td><strong>Write(B)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Unlock(A)</strong></td>
<td><strong>Unlock(B)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Unlock(B)</strong></td>
<td></td>
</tr>
</tbody>
</table>

Output: 950, 2050, 3000
Venn Diagram for Schedules

- Serial
- View Serializable
- Conflict Serializable
- All Schedules
- Serializable
- View Serializable
- Conflict Serializable
- Serial
- Avoid
- Cascading
- Aborts
Q: Which schedules does Strict 2PL allow?

- Serializable
- View Serializable
- Conflict Serializable
- All Schedules

Avoid Cascading Aborts

Serial
Another Venn diagram

- Serializable schedules
- 2PLC
- 2PL schedules
- Serial sch’s
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:  

T1
T2
T4
T3
Deadlock Detection, Part 4

Example:

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

Diagram:

T1  →  T2  
    
T4  

T3  

Diagram:

T1  →  T2  
    
T4  

T3  

Deadlock Detection, Part 9

Example:

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

Diagram: [Diagram showing a directed graph with nodes T1, T2, T3, and T4 connected by arrows]
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)      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:  S(D), S(C),    X(B)  

Diagram: T1 -> T2, T4 -> T3
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?

T1 → T2 → T3 ↘ T4 → T1
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

**Wait-Die**
- Ti wants
- Tj has

**Wound-Wait**
- Ti wants
- Tj has

Priorities
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!
<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT * FROM Product WHERE color='blue'</td>
<td>INSERT INTO Product(name, color) VALUES ('A3','blue')</td>
</tr>
<tr>
<td>SELECT * FROM Product WHERE color='blue'</td>
<td></td>
</tr>
</tbody>
</table>

T1 sees a “phantom” product A3
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

- **SERIALIZABLE** – No phantoms, all reads repeatable, no “dirty” (uncommitted) 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
  - READ WRITE

  isolation  level
  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