Relational Database System Implementation

CS122 – Lecture 18

Winter Term, 2018-2019

Last Time: Transaction Isolation

- Model transactions as a sequence of reads and writes
- A pair of schedules *S* and *S'* are *conflict equivalent* if:
  - One schedule can be transformed into the other, solely by swapping adjacent non-conflicting operations
  - Adjacent operations conflict if they involve the same data item, and at least one operation is a write
- A schedule *S* is *conflict serializable* if it is conflict equivalent to a serial schedule
- Not all conflict serializable schedules maintain atomicity and durability!

Last Time: Transaction Isolation (2)

- This schedule is conflict serializable, but not recoverable
- Problem: T_j reads a value that T_i writes, but wants to commit before T_i commits or aborts.
- A schedule S is *recoverable* if, for every pair of txns T_i and T_j:
  - If T_j reads a data-item previously written by T_i, then T_j is not allowed to commit until T_i first commits

T_i:	T_j:
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
	read(C);
	$C := C + 30$;
	write(C);
	commit.
read(B);	
$B := B + 50$;	
write(B);	
abort.	

Last Time: Transaction Isolation (3)

- If T_i aborts then we must abort T_j too
  - Called a *cascading rollback*
- *Cascadeless schedules* prevent cascading rollbacks
- A schedule S is cascadeless if, for every pair of txns T_i and T_j:
  - If T_j reads a data-item previously written by T_i, then T_j is not allowed to perform this read until T_i first commits

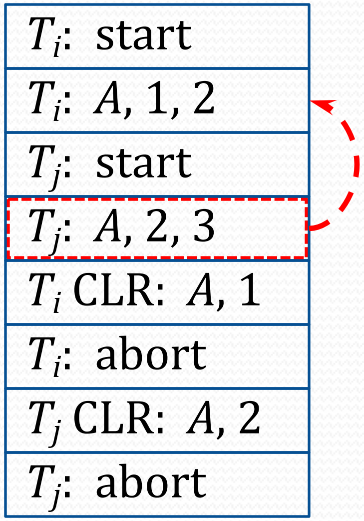
T_i:	T_j:	T_k:
read(A);		
$A := A - 50$;		
write(A);		
	read(A);	
	$A := A - 30$;	
	write(A);	
	read(C);	
	$C := C + 30$;	
	write(C);	
		read(C);
		$C := C * 1.03$;
		write(C);
read(B);		
$B := B + 50$;		
write(B);		
abort.		
	abort.	
		abort.

Last Time: Transaction Isolation (4)

- Write-ahead logging introduces a subtle read-dependency between transactions
    - Previous approaches cannot handle blind writes properly
- To simplify recovery processing, further constrain schedules to be strict
- A schedule S is *strict* if, for every pair of transactions T_i and T_j:
    - If T_j reads <u>or writes</u> a data-item previously written by T_i, then T_j is not allowed allowed to do this until T_i first commits

T_i:	T_j:
$A := 2$	
write(A);	
	$A := 3$
	write(A);
abort.	
	abort.

Write-Ahead Log:



T_i: start
T_i: A, 1, 2
T_j: start
T_j: A, 2, 3
T_i CLR: A, 1
T_i: abort
T_j CLR: A, 2
T_j: abort

Strict Schedules

- Would like our transaction schedules to be *strict*
  - Conflict-equivalent to a serial execution schedule
  - Disallows cascading rollbacks
  - Makes recovery processing very easy

- How do we enforce only strict transaction execution schedules in a multi-user database?

Concurrency Control System

- A *concurrency control* system must govern all operations of all transactions in the database
  - A transaction wants to read or write a data item…
  - The concurrency control system may allow, delay, or even deny the operation
- Conservative schedulers tend to delay operations
  - By delaying operations, scheduler can often reorder them to avoid conflicts
- Aggressive schedulers tend to perform operations immediately
  - Can’t reorder operations once they are performed…
  - Sometimes run into unresolvable conflicts that require aborting a transaction

Concurrency Control System (2)

- Several different ways to implement concurrency control, with different characteristics
- Conflict-serializable schedules:
  - Allow adjacent operations of two transactions to be swapped when they don’t conflict
  - Two adjacent operations conflict when:
    - Both operations are on the same data-item
    - At least one of the operations is a write
- A simple idea for implementing concurrency control:
  - Use locks on data-items to enforce concurrency control
  - If two transactions perform conflicting operations, locks will simply disallow reordering the operations

Lock-Based Protocol

- Reads don't conflict with other reads, but writes conflict with everything…
- Introduce two kinds of locks:
  - A *shared-mode* lock acquired by readers
    - Multiple transactions can hold a shared-mode lock on a single data item
  - An *exclusive-mode* lock acquired by writers
    - Only one transaction can hold an exclusive-mode lock on a data-item
- A *lock compatibility function* specifies when different lock modes are compatible:

	shared	**exclusive**
shared	true	false
exclusive	false	false

Lock-Based Protocol (2)

- Introduce operations for transactions to use:

lock-S(Q)	Acquire a shared lock on data-item Q
lock-X(Q)	Acquire an exclusive lock on data-item Q
unlock(Q)	Release a lock on data-item Q

- A *lock manager* is responsible for handling requests
- Transactions must guard reads and writes with lock/unlock operations
- Next operation in transaction *cannot* be performed until lock is granted

T_i: lock-X(A);
read(A);
$A := A - 50$;
write(A);
unlock(A);
commit.

Lock Manager

- The Lock Manager handles requests for locks
  - Must keep track of which transactions hold which locks

- If a request can be satisfied, the Lock Manager grants the lock to the requester immediately
- If a request is blocked by an existing lock, the Lock Manager blocks requester until lock becomes available

Lock Manager (2)

- Lock manager keeps a mapping of all currently locked data items, along with lock-holders and requesters
  - Often called a *lock table*
- Also helpful to keep a mapping of active transactions, and all locks and requests held by each transaction
  - Makes it easy to release all locks at commit or abort time
  - When a transaction is aborted, must also clear out its lock requests

Lock Manager (3)

- When a lock request arrives:
  - If the data item is not currently locked, lock manager can grant it immediately, regardless of lock mode
- If the data item is already locked by a transaction:
  - Lock manager must ensure that the new lock-request is compatible with mode of the current lock
  - If so, lock manager can generally grant the request immediately (with caveats)
  - Otherwise, the lock-request is added to a request-queue for that data item

Lock Manager (4)

- Lock manager must prevent *starvation*
- Example: data item Q
  - T_1 requests a shared lock on Q; granted immediately
  - T_2 requests an exclusive lock on Q; T_2 must wait
  - T_3 requests a shared lock on Q; granted immediately
  - T_1 releases its lock on Q
  - T_4 requests a shared lock on Q; granted immediately
  - T_3 releases its lock on Q
  - …
- If we *always* grant compatible requests, some transactions may never receive their requested locks

Lock Manager (5)

- To prevent starvation, only grant incoming request if:
  - Request is compatible with current lock mode
  - There is no earlier lock-request still waiting for the lock
- When an unlock request arrives:
  - Lock manager removes lock entry for the unlocking txn
  - If other transactions are waiting to lock the data item, handle those requests as previously specified
    - e.g. a single exclusive-mode lock request may be granted, or a series of shared-mode lock requests may be granted
  - An unlock operation from one transaction may unblock another transaction, allowing it to resume its progress

Locking and Scheduling

- Is wrapping individual reads and updates with locks sufficient to enforce conflict-serializable schedules?
- Example:
  - T_i transfers \$30 from B to A
  - T_j retrieves sum of balances
- No! Conflicting operations may still be swapped.
  - If all of T_j executes between T_i's unlock(A) and lock-X(B) steps, T_j's result will be wrong

T_i:

```
lock-X($B$);
read($B$);
$B := B – 30$;
write($B$);
unlock($B$);
lock-X($A$);
read($A$);
$A := A + 30$;
write($A$);
unlock($A$);
commit.
```

T_j:

```
lock-S($A$);
read($A$);
unlock($A$);
lock-S($B$);
read($B$);
unlock($B$);
display($A + B$);
commit.
```

Locking and Scheduling (2)

- Must specify rules governing when transactions are allowed to lock and unlock data items
  - Called a *locking protocol*
- Locking protocol restricts the set of allowed schedules
  - A schedule *S* is *legal* under a given locking protocol, if *S* follows the locking rules specified by the protocol
- Goal:
  - Design the locking protocol so that we are restricted to only conflict-serializable (or preferably strict) schedules

Two-Phase Locking Protocol

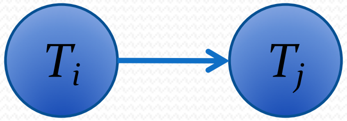
- Require that transactions manage locks in two phases
- *Growing* phase:
  - A txn may acquire new locks, and may not release any lock
- *Shrinking* phase:
  - A txn may release locks, and may not acquire any new locks
- Transactions start in the growing phase
  - As transaction operates on various data items, it acquires locks on those items
- Once a txn releases any lock, it enters the shrinking phase
  - It can only release locks, until all of its locks are released
- Called the *two-phase locking* protocol (2PL for short)

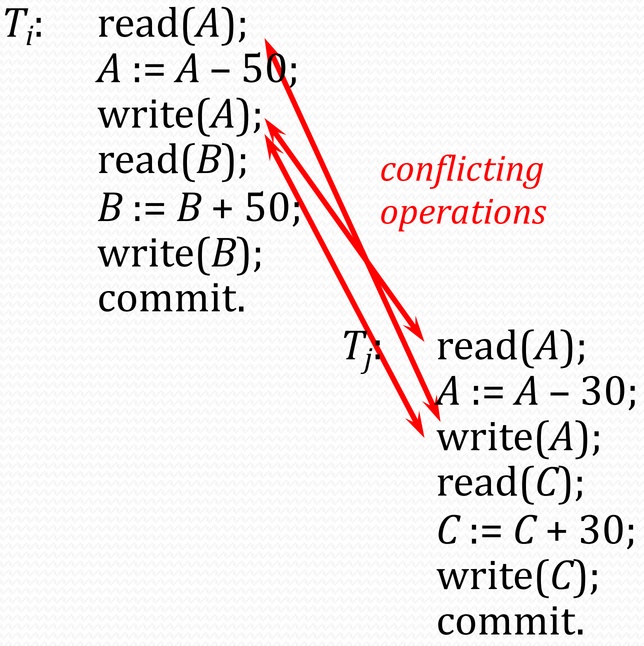
Two-Phase Locking Protocol (2)

- The two-phase locking protocol enforces conflict-serializable transaction schedules…
- To prove this, we need a way of reasoning about transaction schedules
- Define a *precedence graph* of all transactions participating in a schedule S
  - Also known as a *serialization graph*
- Vertices in precedence graph are the transactions in S
- Edges in graph are edges $T_i \rightarrow T_j$, such that T_i performs a conflicting operation before T_j does, in the schedule S

Precedence Graph

- Vertices in precedence graph are the transactions in S
- Edges in graph are edges $T_i \rightarrow T_j$, such that T_i performs a conflicting operation before T_j does, in the schedule S
- Example: a serial execution schedule
- Precedence graph:




T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B);
commit.

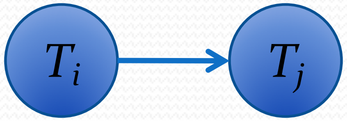
conflicting operations

T_j: read(A);
$A := A - 30$;
write(A);
read(C);
$C := C + 30$;
write(C);
commit.

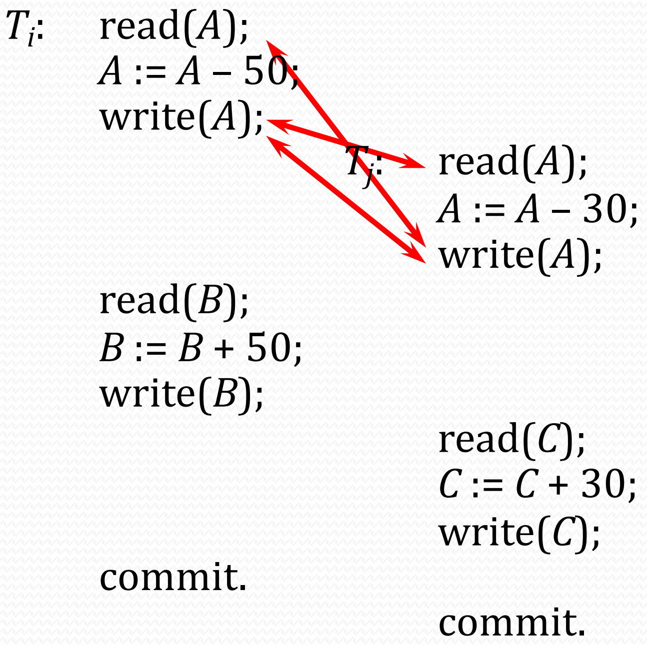
- Which operations conflict?
- Only one arrow, from T_i to T_j
  - All operations in T_i that conflict with ones in T_j are performed *before* the conflicting ones in T_j

Precedence Graph (2)

- Another example: a serializable execution schedule
- Which operations conflict?
- Precedence graph:



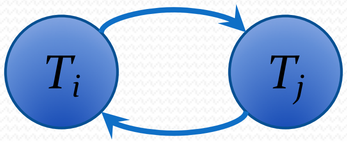
- Again, only one arrow, from T_i to T_j
  - <u>All</u> operations in T_i that conflict with ones in T_j are performed *before* the conflicting ones in T_j

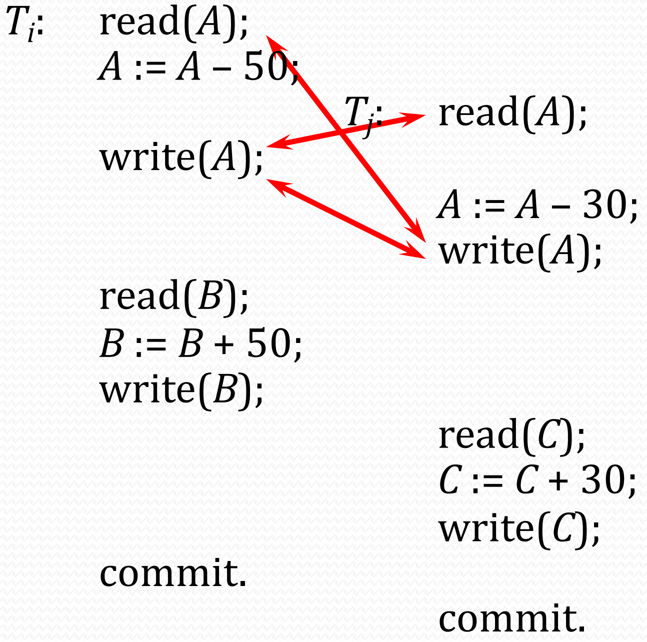


T_i:	T_j:
read(A);	
$A := A - 50$;	
write(A);	
	read(A);
	$A := A - 30$;
	write(A);
read(B);	
$B := B + 50$;	
write(B);	
	read(C);
	$C := C + 30$;
	write(C);
commit.	
	commit.

Precedence Graph (3)

- One more example: a non-serializable schedule
  - Clearly produces spurious results
- Now, precedence graph has two arrows





T_i:	T_j:
read(A); $A := A - 50$;	
	read(A);
write(A);	
	$A := A - 30$; write(A);

read(B);
$B := B + 50$;
write(B);	

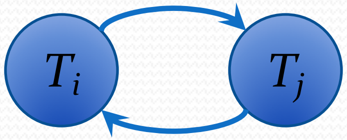
	read(C);
	$C := C + 30$;
	write(C);

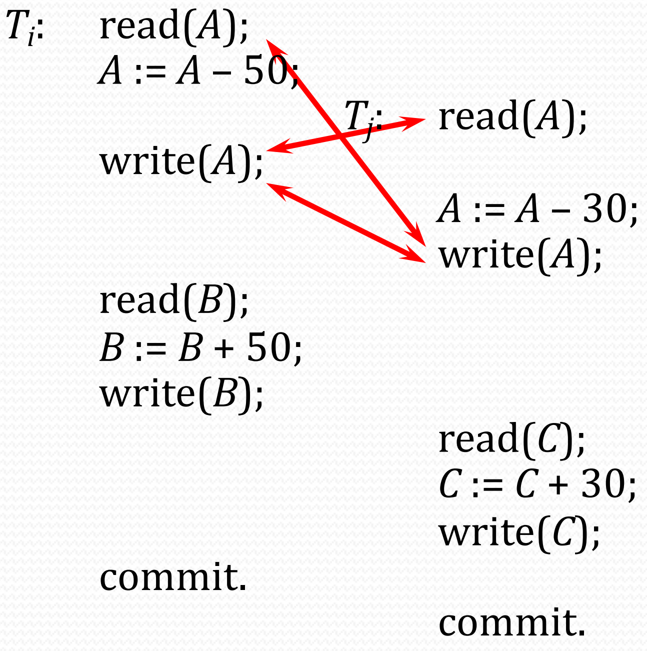
commit.	
	commit.

  - T_i reads A before T_j writes A
  - T_i writes A before T_j writes A
  - T_j reads A before T_i writes A

Precedence Graph (3)

- A cycle in the precedence graph indicates that the schedule is <u>not</u> serializable
- Cycle indicates that two txns in the schedule have conflicting operations that are interleaved
- <u>Cannot</u> swap these conflicting operations to get to a serial schedule…
  - <u>Not</u> equivalent to a serial schedule





T_i:	T_j:
read(A); $A := A - 50$;	
	read(A);
write(A);	
	$A := A - 30$; write(A);

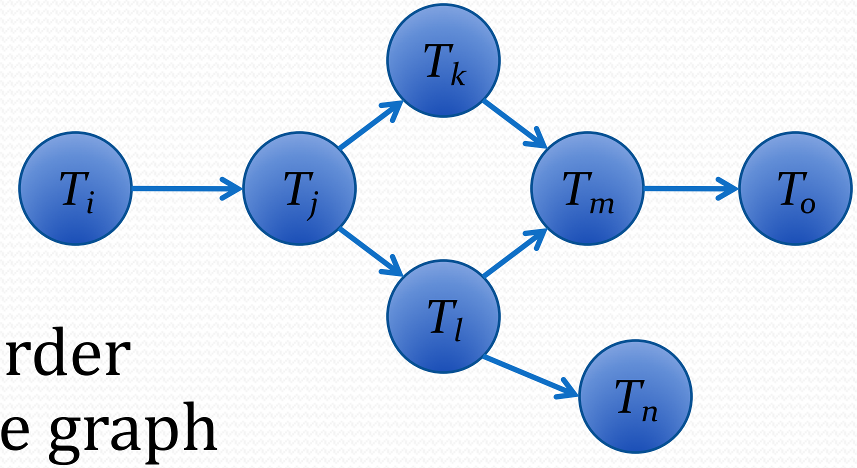
read(B);
$B := B + 50$;
write(B);	

	read(C);
	$C := C + 30$;
	write(C);

commit.	
	commit.

Precedence Graph (4)

- Can certainly have precedence graphs with more interesting structures
- As long as graph has no cycles, it represents a serializable schedule
- Graph imposes a partial order over all transactions in the graph



- Any linear order consistent with the partial order specified by the graph is called a *serializability order*
  - Indicates the schedule is equivalent to a serial execution of transactions in the serializability order

2PL and Serializability

- If 2PL doesn't allow cycles in the precedence graph, then it will only allow conflict-serializable schedules
- In two-phase locking, every transaction has a *lock point*
  - The point in the transaction's execution when it acquires its last lock
  - At that point, the txn holds all locks it will ever acquire
- A schedule can only perform one operation at a time
  - Every lock request and release occurs at a different time
- Every transaction's lock point is distinct

2PL and Serializability (2)

- If T_i → T_j in the precedence graph:
  - T_i performed *some* operation that conflicted with an operation in T_j (e.g. on data item Q), before T_j's operation
  - Before T_i could perform this operation on Q, it had to lock Q. Similarly, T_j must lock Q before doing its thing.
  - Therefore, T_i had to release its lock on Q before T_j could acquire its lock on Q
- To follow two-phase rule, T_i has to enter the shrinking phase before T_j can acquire the lock
  - T_i's lock point occurs before T_j's lock point

2PL and Serializability (3)

- If $T_i \rightarrow T_j \rightarrow T_k$ in the precedence graph:
  - As before, T_i released a lock before T_j acquires its lock
  - Similarly, T_j released a lock before T_k acquires its lock
  - T_j is in the shrinking phase before T_k acquires its lock

- Transactions that follow two-phase locking can be ordered by their lock points
- Can extend this to arbitrary chains of transactions using induction

2PL and Serializability (4)

- Finally, assume we have T_1 → T_2 → … → T_n → T_1
    - A cycle in precedence graph; not a serializable schedule
- To arrive at this situation:
    - T_1 released some lock before T_2 could acquire its lock
    - T_2 released some lock before T_3 could acquire its lock
    - …
    - T_n released some lock before T_1 could acquire its lock
- This situation can only occur if T_1 tries to acquire a lock <u>after</u> it has already released a lock…
- This is <u>disallowed</u> by the two-phase locking protocol!

2PL and Serializability (5)

- Two-phase locking protocol only allows conflict-serializable execution schedules
- Transactions can be ordered based on their lock points
- This ordering is a serializability order for the entire set of transactions
    - The 2PL schedule is equivalent to a serial schedule where txns are executed in order of their lock points

Two-Phase Locking Example

- Previous example, updated to follow two-phase rule:
  - Now we know it is conflict-serializable
- What new problem do we have?
  - Shared and exclusive locks are incompatible…
- A schedule executing these transactions is prone to deadlock!

T_i:

```
lock-X(B);
read(B);
B := B – 30;
write(B);
lock-X(A);
read(A);
A := A + 30;
write(A);
unlock(B);
unlock(A);
commit.
```

T_j:

```
lock-S(A);
read(A);
lock-S(B);
read(B);
unlock(A);
unlock(B);
display(A + B);
commit.
```

2PL and Deadlocks

- A two-phase locking schedule that deadlocks:
- Can’t avoid this issue…
  - Never know what data items a transaction might use!
- Only recourse is to identify deadlocks when they occur
  - Choose one transaction in the deadlock, and abort it.
  - Aborted transaction is called the *victim*

T_i:	T_j:
lock-X(B);	
read(B);	
$B := B - 30$;	
write(B);	
	lock-S(A);
	read(A);
lock-X(A); WAIT	
read(A);	lock-S(B); WAIT
$A := A + 30$;	read(B);
write(A);	unlock(A);
unlock(B);	unlock(B);
unlock(A);	display($A + B$);
commit.	commit.

2PL: Detecting Deadlocks

- Current Lock Manager design:
  - Lock manager tracks every data item that is locked
    - Lock manager records the transaction that has the item locked, and the lock mode (shared or exclusive)
  - If other transactions are waiting to lock a data item, the lock manager also records these lock-requests
- The lock manager also maintains a *waits-for graph*, tracking relationships between waiting transactions
  - If a transaction T_i holds a lock on a data item Q, and T_j is waiting to lock Q, the waits-for graph records $T_j \rightarrow T_i$

2PL: Detecting Deadlocks (2)

- If waits-for graph contains a cycle, a deadlock exists!
  - All transactions in the cycle are deadlocked, not just one
- How many outgoing edges will a transaction have in the waits-for graph?
  - Depends on the mode of the current lock on the item!
  - e.g. if item is locked in shared-mode by multiple txns, and an exclusive-mode request is made, requester will have outgoing edges to all txns holding the lock
- Multiple deadlock cycles could exist in waits-for graph
  - One transaction could be involved in multiple cycles
  - Deadlock detection must identify all cycles in graph

2PL: Detecting Deadlocks (3)

- Waits-for graph can be updated every time a request cannot be granted immediately
  - If a request can be granted immediately, no reason to update the waits-for graph… transaction isn't waiting…
- When a transaction unlocks a data item, one or more waiting requests can be granted
  - Must again update the waits-for graph
- When a txn aborts, all of its locks and outstanding requests are removed from the lock manager
  - Again, must update the waits-for graph

2PL: Detecting Deadlocks (4)

- When should deadlock detection be invoked?
  - Will certainly consume CPU resources, so don't want to run it all the time
- Don't need to run it all the time…
  - Deadlocks have a nice property: they don't go away!
- Only need to consider running deadlock detection when a lock request can't be granted right away
  - e.g. if a lock request isn't satisfied within a specific time interval, invoke deadlock detection algorithm

2PL: Resolving Deadlocks

- If deadlock is detected, another important question:
- How should we choose a victim transaction to abort?
- Example:
  - Transaction T_1 is performing a long-running analysis
  - Transaction T_2 involves three quick operations
  - If T_1 and T_2 deadlock, which should be aborted?
  - Preferably, T_2 should be aborted so that less work is lost
- Goal:
  - Choose a victim to abort that will incur the least cost

2PL: Resolving Deadlocks (2)

- Identifying victim that will incur least cost is difficult to do
- Can consider definite measures:
  - How long each transaction in the deadlock cycle has been running
    - Abort the youngest transaction in the cycle?
  - How costly the transaction itself will be to abort:
    - How many data-items has the transaction modified?
    - The more writes the transaction has performed, the more costly it will be to rollback all changes
  - How many deadlock cycles the transaction is involved in
    - Every deadlock cycle must be broken! If multiple cycles can be broken by aborting one transaction, everybody [else] wins.

2PL: Resolving Deadlocks (3)

- Can also try to predict the future:
  - How close is each transaction to being finished?
    - If not throwing away a large amount of work, would be nice to abort transactions that still have a long way to go
  - How many more data items will the transaction need?
    - Prefer to abort a transaction that requires more resources over one that requires less
  - Can be challenging to make these predictions, but the set of queries against a DB usually doesn't vary a lot
    - Can observe past behavior of queries
- Or, just pick one randomly ☺

Two-Phase Locking Protocol

- So far, two-phase locking protocol ensures conflict-serializable execution schedules…
- …but we really wanted strict schedules.
  - Rule out cascading aborts, nonrecoverable schedules, and complicated recovery processing

- This form of 2PL is called <u>basic</u> two-phase locking
- Next time, discuss refinements of two-phase locking with much more desirable characteristics