Transaction Management

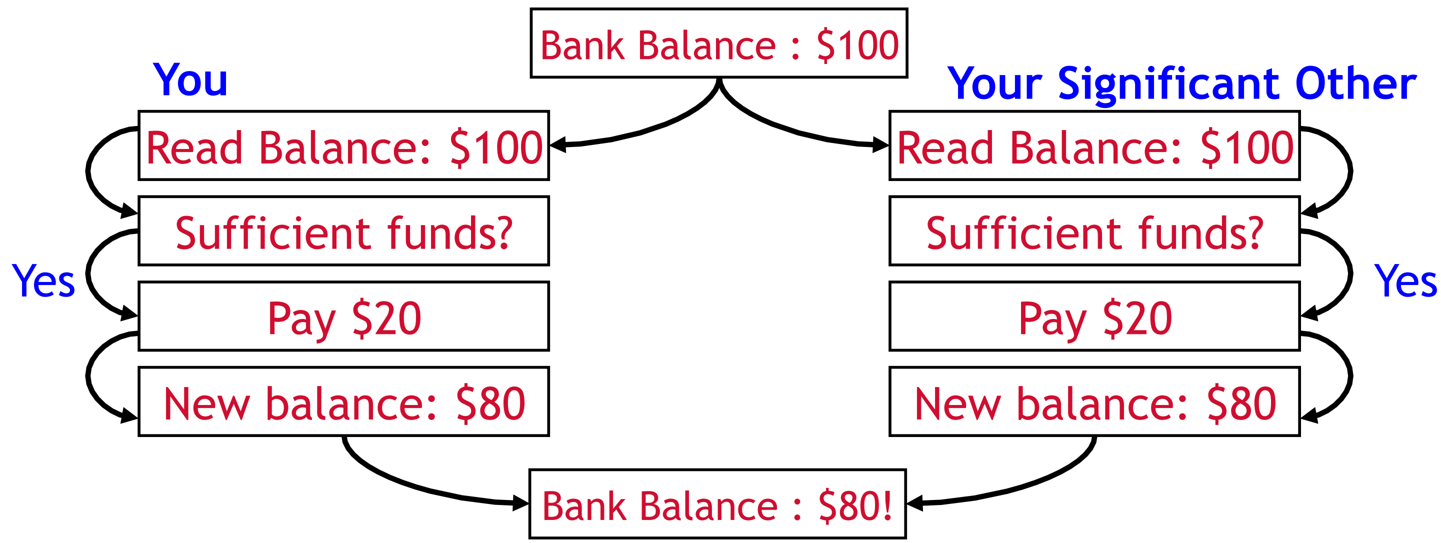
Chapter 16 (except 16.6),
Chapter 17 (except 17.5 and beyond)

Transactions

- Foundation for concurrent execution and recovery in DBMS
- Transaction is an atomic unit of work
  - E.g., Debit $500 from my bank account
- Transaction consists of multiple actions
- For performance, DBMS can interleave actions from different transaction
- Must guarantee same result as executing transactions serially

Example 1

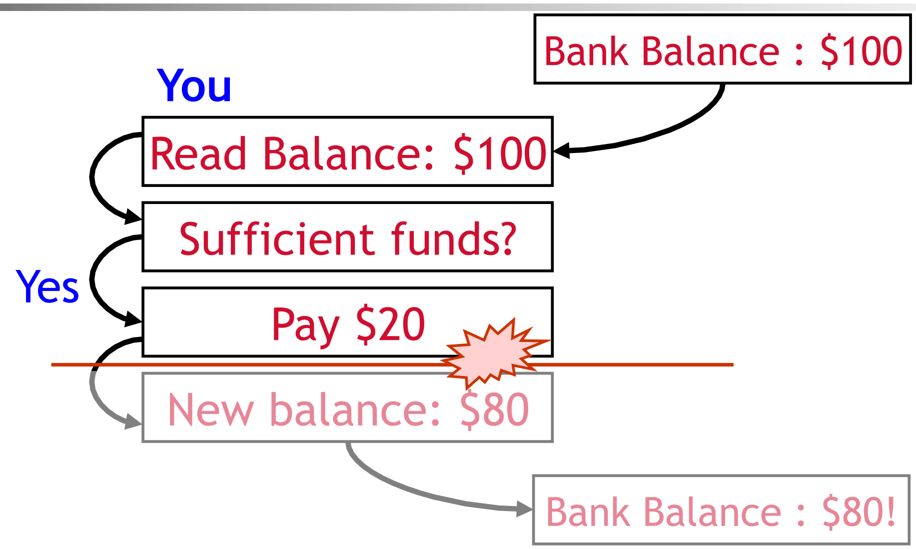
Read (A);
Check (A > $20);
Pay ($20);
A = A - 20;
Write (A);



- Interleaving actions of different transactions can cause inconsistency
- DBMS should provide users an illusion of a single-user system
- Could insist on admitting only one transaction at a time
  - Lower utilization: CPU / IO overlap
  - Long running queries starve other queries, reduce overall response time

Example 2

```
Read (A);
Check (A > $20);
Pay ($20);
A = A - 20;
Write (A);
```



DBMS must also guarantee that changes made by partially completed transactions are not seen by other transactions

The ACID Properties

TM
Trans. Mgmt.
(logging)

- **A**tomicity: All or nothing property
  - Xacts do not execute partially

User

- **C**onsistency:
  - Consistent DB + Xact $\Rightarrow$ consistent DB
  - We assume that each submitted transaction, if executed, satisfies any integrity constraints

CC
Concurrency Ctrl.
(locking)

- **I**solation: Each Xact appears to execute without concurrent Xacts, i.e., serially.

  => Any parallel execution should be equivalent to a serial execution

RM
Recovery Mgmt.
(WAL, …)

- **D**urability: If a Xact commits, its effects persist.

Schedules

- Transaction: a list of actions
  - Read(X), Write(X), commit, abort
- Schedule: An interleaving of the actions from a set of transactions
  - Complete Schedule: Each transaction ends in commit or abort
- Initial State + Schedule = Final State

Time

T1	T2
begin	
R(A)	
W(A)	
	begin
	R(B)
	W(B)
R(C)	
W(C)	
	commit
abort	

Serial Schedule

- Serial Schedule:
  - No interleaving actions among transactions
  - Automatically isolated

Non-serial

T1	**T2**
begin	
R(A)	
W(A)	
	begin
	R(B)
	W(B)
R(C)	
W(C)	
	commit
commit	

serial

T1	**T2**
	begin
	R(B)
	W(B)
	commit
begin	
R(A)	
W(A)	
R(C)	
W(C)	
commit	

Serializable Schedule

- A serializable schedule is equivalent to a serial schedule
  - Same result
- The two schedules on the left are both serializable

Time

Non-serial

T1	T2
begin	
R(A)	
W(A)	
	begin
	R(B)
	W(B)
R(C)	
W(C)	
	commit
commit	

serial

T1	T2
	begin
	R(B)
	W(B)
	commit
begin	
R(A)	
W(A)	
R(C)	
W(C)	
commit	

Serializable schedule is essential

- For a schedule to be acceptable to a database, it must be serializable
  - Guarantees ACID properties
- Can different serial schedules have different final states?
  - Yes, all acceptable
- Aborted Xacts?
    - We will try to make them ‘disappear’ by undoing their effect
- Other external actions (besides R/W to DB)
    - e.g. print a computed value, fire a missile, …
    - Assume (for this class) these values are written to the DB, and can be undone

Example

- Given T1, T2.
- The two schedules on the left are both acceptable for executing T1 and T2.
- They produce different results, but both serializable and thus OK

Time

T1	T2
begin	
R(A)	
W(A)	
	begin
	R(A)
	W(A)
R(C)	
W(C)	
commit	
	commit

T1	T2
	begin
	R(A)
	W(A)
	commit
begin	
R(A)	
W(A)	
R(C)	
W(C)	
commit	

Equivalent to T1,T2

Anomalies due to Interleaving

- Two actions on the same data “object” conflict if at least one is a write()
- Three anomalous situations for transactions T1 and T2
  - Write-read (WR) conflict
  - Read-write (RW) conflict
  - Write-write (WW) conflict

WR Conflict

- "Dirty read"
- Could lead to a non-serializable execution
- Example:
  - @Start (A,B) = (1000, 100)
    - End (990, 210)
  - T1→T2:
    - (900, 200) → (990, 220)
  - T2→T1:
    - (1100, 110) → (1000, 210)

Database Inconsistent

T1: Transfer $100 from A to B	*T2: Add 10% interest to A & B*
begin	
	begin
R(A), reads 1000	
A = A-100, **W(A)**	
	R(A)
	A *= 1.1, **W(A)**
	R(B), reads 100
	B *= 1.1, **W(B)**
	commit
R(B)	
B += 100, **W(B)**	
commit	

RW Conflicts

- "Unrepeatable read"
- T1 reads the value of object A, then T2 updates A (before T1 has committed)

T1	T2
R(A) = 3	
W(B) : B=A	
	W(A) = 5
	Commit
R(A) = 5	
W(A) : A=A+1	
Commit	

Why isn't this schedule serializable?

WW Conflict

- Overwriting uncommitted data
- T2 overwrites what T1 wrote
- Usually occurs in conjunction with other anomalies
  - Unless you have "blind writes"

Students in same group (A and B) should get the same project grade

T1 (GSI)	T2 (Prof)
W(A) = 1	
	W(A) = 2
	W(B) = 2
W(B) = 1	
Commit	
	Commit

Why isn't this schedule serializable?

What about schedules with aborts?

- Acceptable schedule if equivalent to a serializable schedule of *committed* transactions
- As if aborted transactions *never happened*

T1 (GSI)	T2 (Prof)
	W(A) = 2
	W(B) = 2
W(A) = 1	
W(B) = 1	
Abort	
	Commit

Cascaded Aborts

- How does one undo the effects of a transaction T1?
  - Covered in logging and recovery lecture
- What if another transaction T2 sees these effects??
  - Must abort and undo T2 too!
  - (Called cascading abort)

Recoverable Schedule

- Can we undo effects of aborted transactions?
  - Not always

Example:

T1 reads decrements account balance by $100

T2 adds 5% cashback

This schedule is NOT recoverable.

T1	T2
R(balance) = 1000	
W(balance) = 1000 - 100	
	R(balance) = 900
	W(balance) = 900 * 1.05
	Commit
Abort	

Goals for Acceptable schedules

- Must be serializable
- Additional desirable goals:
  - Recoverable schedule - Transactions commit only after all transactions whose changes they have read commit.
  - Avoid cascading aborts (ACA) - Transactions read only the changes of committed transactions.

Transaction and Constraints

Create Table A (<u>akey</u>, *bref*, …)

Create Table B (<u>bkey</u>, *aref*, …)

Q: How to insert the first tuple, either in A or B?

- Solution:
  - Insert tuples in the same transaction
  - Defer the constraint checking
- SQL constraint modes
  - BEGIN DEFERRED: Check at commit time.
  - BEGIN IMMEDIATE: Check immediately

Intro to Locking

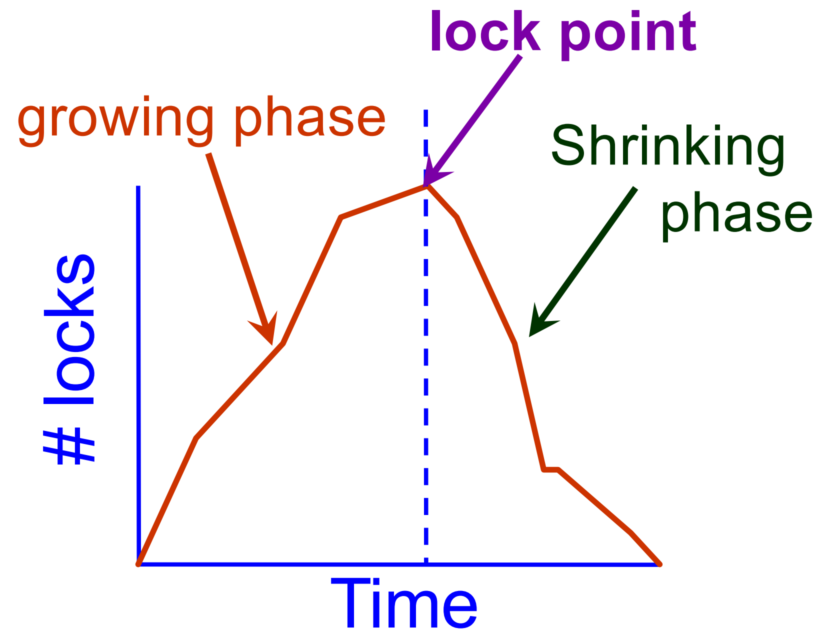
- What can a DBMS do to guarantee an acceptable schedule?
- Lock info maintained by a "lock manager":
  - Stores (XID, RID, Mode) triples.
    - This is a simplistic view; suffices for now.
  - Mode ∈ {Shared,eXlusive} or {Read,Write} lock
  - Lock compatibility table:

	--	R	W
--	√	√	√
R	√	√	
W	√		

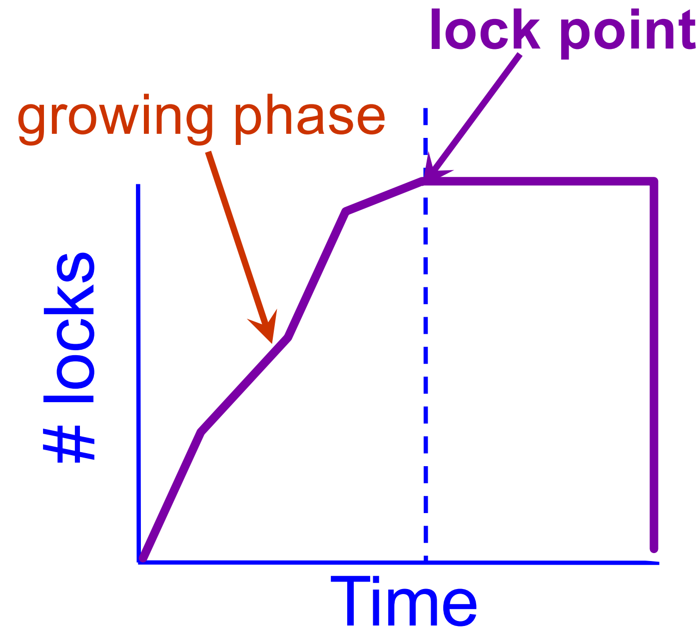
- If a transaction can't get a lock
  - Suspended on a **wait queue**

Two-Phase Locking (2PL)

- **2PL**:
  - If T wants to read (modify) an object, first obtains a READ (WRITE) lock
  - If T releases any lock, it can acquire *no new locks*!
  - Gurantees serializability!



- **Strict 2PL**:
  - Hold all locks until end of Xact
  - Guarantees serializability and ACA too!
    - Note ACA schedules are always recoverable



Example

- Two accounts, A and B
- Two transactions, T1 and T2
- T1: Transfer $100 from A to B
- T2: Add 10% interest to A and B

Example - Strict 2PL

T1 acquires READ lock on A

T1 acquires WRITE lock on A

T1 acquires READ lock on B

T1 acquires WRITE lock on B

T1 releases all locks

T2 acquires READ lock on A

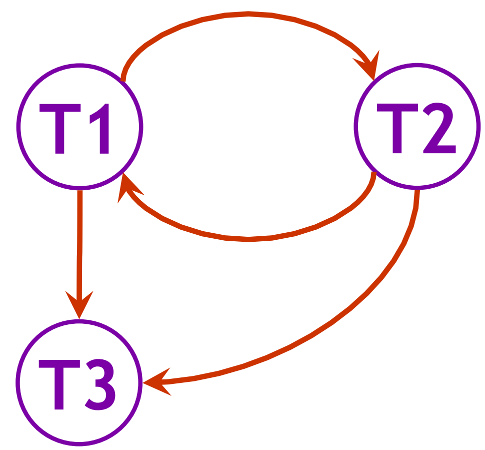
…

T1: Transfer \$100 from A to B	*T2: Add 10% interest to A & B*
begin	
	begin
R(A); A -= 100	
W(A)	
R(B); B+= 100	**Wait**
W(B)	
commit	
	R(A); A *= 1.1
	W(A)
	R(B); B *= 1.1
	W(B)
	commit

Precedence Graph

- Precedence (or serializability) graph for schedule L
  - A node for each committed transaction in L
  - An arc from T_i to T_j if some action in T_i precedes and conflicts with some action in T_j

T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit



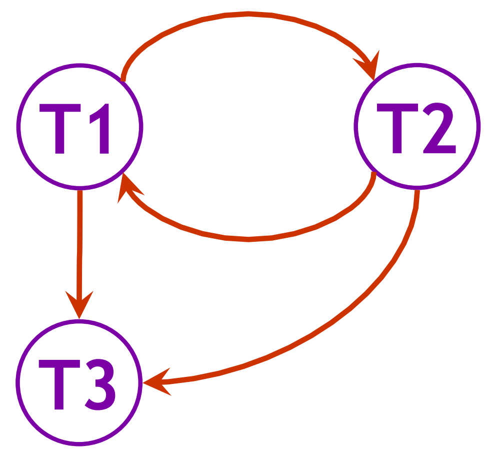
Terminology

- Two schedules are *conflict equivalent* if
  - involve the same transactions
  - order each pair of conflicting transactions in the same way
- A schedule is *conflict serializable* if it is conflict equivalent to a serial schedule
- All conflict serializable schedules are also serializable (opposite is not true!)

Example

- Is this schedule serializable? Is it conflict-serializable too?

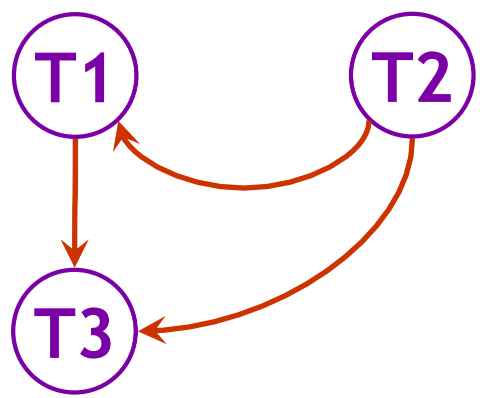
T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit



Conflict Serializability & Graphs

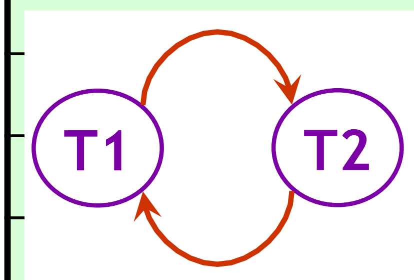
- Theorem: A schedule is conflict serializable if and only if its precedence graph is acyclic
  - Equivalent serial schedule is given by any topological sort over graph

T1	T2	T3
	W(A)	
	Commit	
R(A)		
W(A)		
Commit		
		W(A)
		Commit

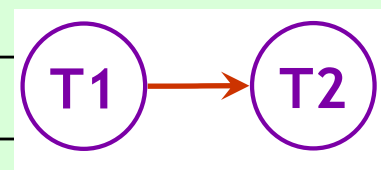


Example

T1: Transfer $100 from A to B	*T2: Add 10% interest to A & B*
begin	
	begin
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	commit
R(B)	
W(B)	
commit	



T1: Transfer $100 from A to B	*T2: Add 10% interest to A & B*
begin	
	begin
R(A)	
W(A)	
R(B)	Wait
W(B)	
commit	
	R(A)
	W(A)
	R(B)
	W(B)
	commit



2PL & Serializability

- 2PL guarantees acyclic precedence graph
  - Guarantees a conflict serializable schedule
  - Intuitively, equivalent serial schedule given by order in which transactions enter shrinking phase
- Why Strict 2PL?
  - Guarantees ACA (read only committed values)
  - How? Hold WRITE locks until end of transaction

Exercise

- Is this schedule allowed by 2PL? Strict 2PL?
- Is it serializable?
  - If so, what is the corresponding serial schedule?
- Is the schedule recoverable? ACA?

T1: Transfer $100 from A to B	*T2: Add 10% interest to A, B & C*
begin	
	begin
	R(C)
	W(C)
R(A)	
W(A)	
R(B)	
W(B)	
commit	
	R(A)
	W(A)
	R(B)
	W(B)
	commit

Scheduling Transactions

- A transaction can be in a state of:
  - Running
  - On a runnable queue, waiting for CPU
  - On a lock queue, waiting for a lock
  - Suspended, waiting for I/O device
- Transaction moves between the running state or one of the queues.

Lock Manager Implementation

- Lock Table: A hash table of Lock Entries. Each entry:
  - OID: object ID
  - Number of transactions holding a lock on this object
  - Mode: shared (READ) or exclusive (WRITE) lock
  - List: A queue of other transactions waiting for a lock on this object

On lock requests for object OID:

- If READ lock requested:
  - If its queue is empty and currently not in WRITE mode:
    - Grant the READ lock, set mode to shared and increment the count
- If WRITE lock requested
  - If no one is currently holding the lock (=> queue will be empty too)
    - Grant the WRITE lock, set mode to exclusive, set count to 1
- Otherwise, add this transaction to the queue and suspend it

Lock Manager Implementation

- On lock release (OID, XID): happens upon commit/abort
  - Update the lock entry
  - Examine lock's wait queue, grant lock to the head of the queue (or to multiple of them if in shared mode)
- Note: Lock request handled atomically! (via mutex in OS)

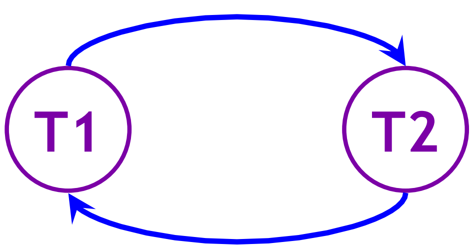
What about deadlocks?

T1 gets READ lock, then WRITE lock on A

T2 gets READ, then WRITE lock on B

T1 waits for READ lock on B

T2 waits for READ lock on A



"Waits-for" Graph

T1: Transfer $100 from A to B	*T2: Add 10% interest to A & B*
begin	
	begin
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	
commit	
	R(A)
	W(A)
	commit

Deadlock Detection

- Observation:
  - Deadlocks are rare
  - Often involve only a few transactions
  - Detect rather than prevent
- Lock Mgr maintains waits-for graph
- Periodically check graph for cycles.
  Abort **some** transaction to break the cycle.
- Simpler hack: *time-outs*.
  Abort if no progress made for a while.

Deadlock Prevention

- Assign priorities to transactions
  - Use timestamp to assign priorities
- Ti requests a lock, held by Tj (in a conflicting mode)
  - Wait-Die: If Ti has higher priority, it waits; else Ti aborts
  - Wound-Wait: If Ti has higher priority, abort Tj; else Ti waits
  - After abort, restart with original timestamp
    - Guarantees the transaction eventually runs!

Non-Locking CC Protocols

- Locking most common technique for guaranteeing transaction serializability
  - Used in most commercial systems
- Other approaches exist (beyond scope of class):
  - Optimistic CC
    - 2PL locking protocols avoid conflict by blocking (waiting)
    - Optimistic CC instead undoes transactions if a conflict occurs
    - Anticipates that conflicts rarely occur
  - Multiversion CC
    - Make sure transactions never have to wait to read an object
    - Maintain multiple versions of each object, each with a timestamp
    - Used by Oracle

Summary

- Locking commonly used by DBMS for concurrency control
- 2PL guarantees a serializable schedule
- Strict 2PL guarantees a serializable, recoverable, ACA schedule
- Lock manager handles lock requests from transactions
- Deadlock rare, but must be detected or prevented

Announcements

- Book Exercises: 16.1, 16.3, 17.5
- Additional Review Exercise: Recall the three kinds of conflicts from last class (RW, WR, WW). For each, think of an example where the conflict occurs. How does 2PL prevent these conflicts?