Transaction Management

Chapter 16 (16.1-16.5),
Chapter 17 (17.1-17.4)

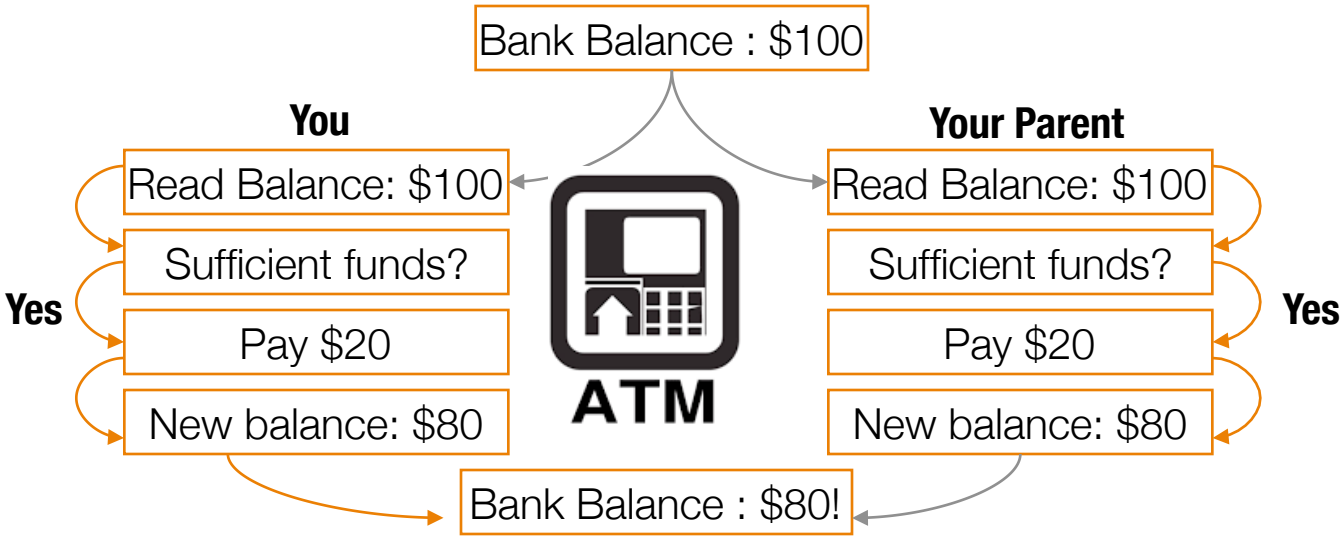
Transactions

- Foundation for concurrent execution and recovery in DBMS
- Transaction is an atomic unit of work
  - e.g. Transfer $500 from chk to saving acct

- Transaction consists of multiple actions
- For performance, DBMS can interleave actions from different transaction
- Must guarantee same result as executing transactions serially

Example 1

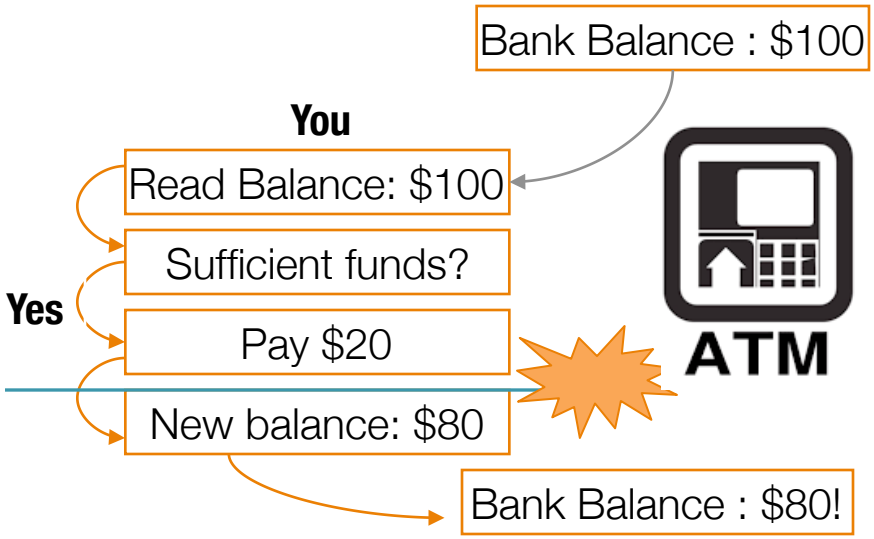
```
Read (A);
Check (A > $20);
Pay ($20);
A = A – 20;
Write (A);
```



- 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

- Atomicity: All actions in the Xact happen, or none happen
  - Don't worry about incomplete Xacts
- Consistency: Consistent DB + consistent Xact ⇒ consistent DB
- Isolation: Execution of one Xact is isolated from that of other Xacts
  - Don't worry about other concurrent Xacts
- Durability: If a Xact commits, its effects persist

Schedules

- Transaction seen as 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
  - Serial Schedule: No interleaving actions among transactions
- Initial State + Schedule = Final State

T1	T2
Begin	
R(A)	
W(A)	
	Begin
	R(B)
	W(B)
R(C)	
W(C)	
	Commit
Abort	

Acceptable Schedules

- Serial schedule is “isolated” and “consistent”
- Serializable schedule:
  - Final state is what some complete serial schedule of committed transactions would have produced.
  - Can different serial schedules have different final states?
    - Yes, and we assume all are OK!
  - Aborted Xacts?
    - Should have no effect whatsoever
  - 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

- Initial State: A=1000, B=100
- Final State: A=990, B=220
- This is serializable because:
  - Running T1 to completion, then running T2 would lead to the same final state (990, 220)
- Note that running T2, then T1 would lead to (1000, 210) which is a different final state. But the schedule is still serializable. Why?

T1: Transfer $100 from A to B	T2: Add 10% interest to A and B
Begin	
	Begin
R(A) /A -=100	
W(A)	
R(B) /B +=100	
W(B)	
	R(A) /A *= 1.1
	W(A)
	R(B) /B *= 1.1
	W(B)
	Commit
Commit	

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 anomaly
  - Read-write (RW) conflict anomaly
  - Write-write (WW) conflict anomaly

WR Conflict Anomaly

- Dirty read: reading an uncommitted change
- 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)

T1: Transfer $100 from A to B	T2: Add 10% interest to A and B
Begin	
	Begin
R(A) /A -=100	
W(A)	
	R(A) /A *= 1.1
	W(A)
	R(B) /B *= 1.1
	W(B)
	Commit
R(B) /B +=100	
W(B)	
Commit	

Inconsistent Database

RW Conflict Anomaly

- "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	

Is this schedule serializable? Why?

WW Conflict Anomaly

- Overwriting uncommitted changes
- 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) = 80	
	W(A) = 95
	W(B) = 95
W(B) = 80	
Commit	
	Commit

Is this schedule serializable? Why?

What About Aborts?

- Serializable schedule: Effect equivalent to a serial 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

What About 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)

Can we always undo effects of aborted transactions?

No! Not all schedules are recoverable

Example:

T1 reads & decrements account balance by $100

T2 adds 5% cashback

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

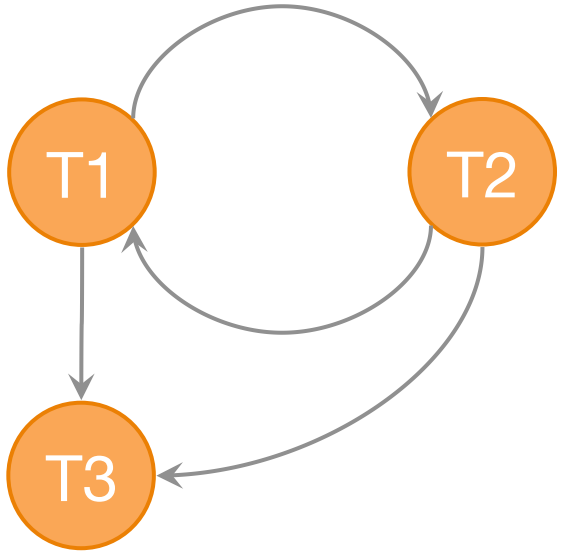
Desirable Goals

- Different goals:
  - Serializable schedule – (Discussed earlier)
  - Recoverable schedule - Transactions commit only after all transactions whose changes they have read commit
    - Note: A recoverable schedule is **not necessarily** serializable and vice versa
  - Avoid cascading aborts (ACA) - Transactions read only the changes of committed transactions
    - ACA ensures Recoverability but Rec. doesn't ensure ACA

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
  - (Remember that a conflict may or may not be an anomaly)

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



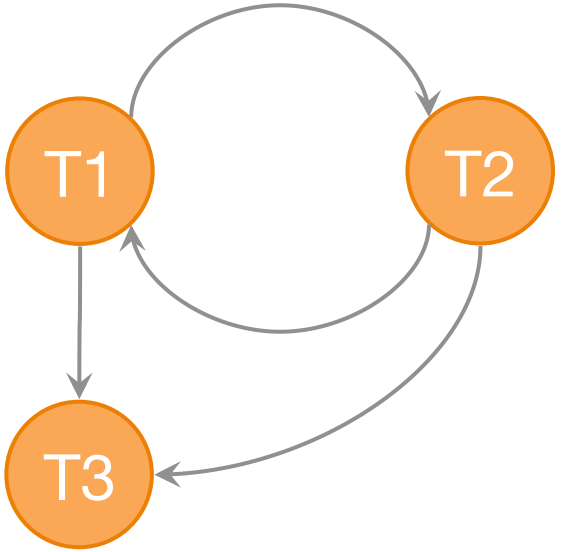
Terminology

- 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?

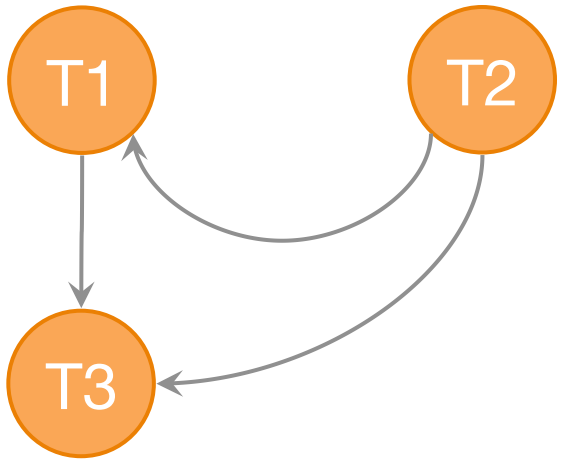
T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit



Conflict Serializability & Graphs

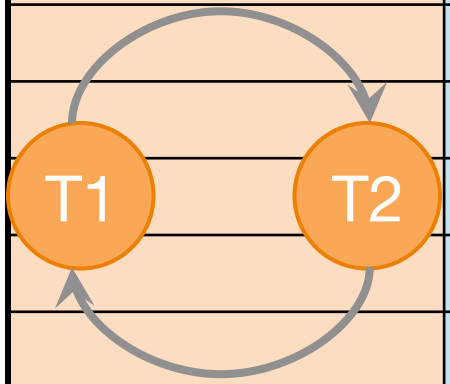
- Theorem: A schedule is conflict serializable if and only if its precedence graph is acyclic
  - Equivalent serial schedule is given by any topological sort over graph

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

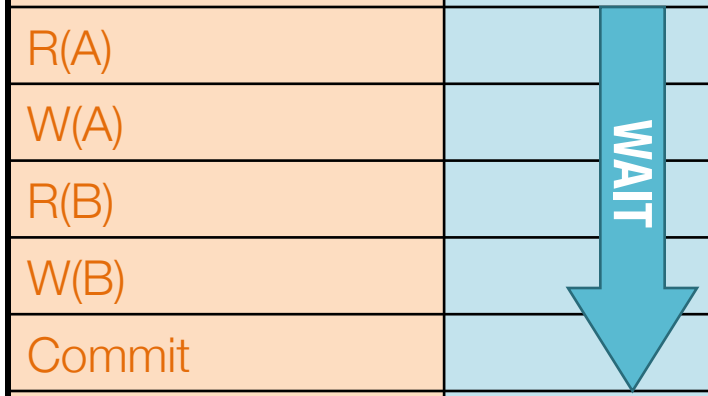


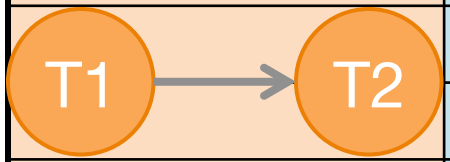
Quiz: Are these schedules conflict-serializable?

T1: Transfer $100 from A to B	T2: Add 10% interest to A and 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 and B
Begin	
	Begin
R(A)	
W(A)	
R(B)	
W(B)	
Commit	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit





Intro to Locking

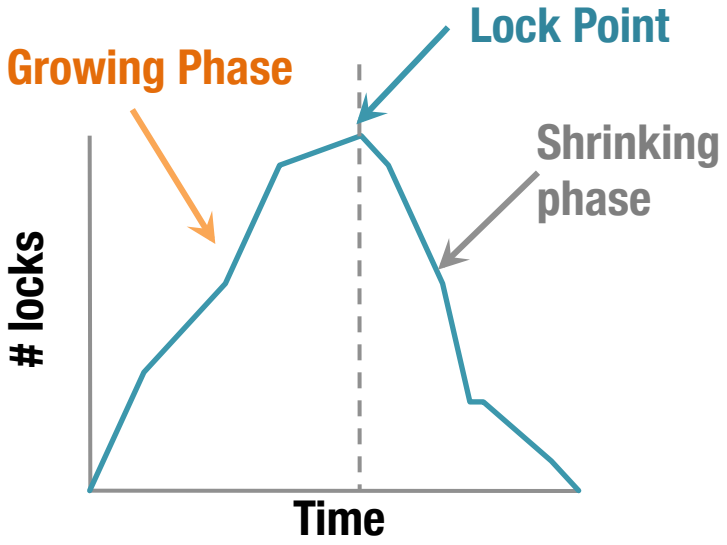
- What can a DBMS do to guarantee an acceptable schedule?
- Lock info maintained by a "lock manager":
  - Stores (XID, RID, Mode) triples.
    - This is a simplistic view; suffices for now.
  - Mode ∈ {S,X}
  - Lock compatibility table:

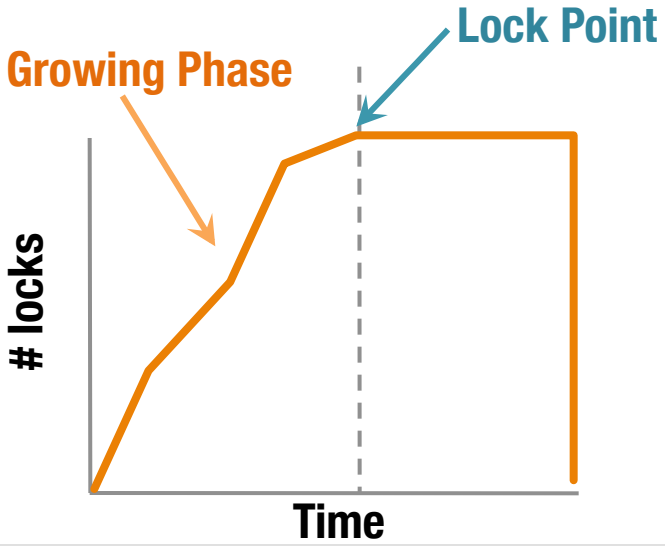
	--	S	X
--	√	√	√
S	√	√	
X	√		

- 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 shared (exclusive) lock
  - If T releases any lock, it can acquire no new locks!
  - Guarantees conflict-serializability (and thus serializability)!
- Strict 2PL:
  - Hold all locks until end of Xact
  - Guarantees conflict-serializability (and thus serializability)
  - Guarantees ACA (and thus recoverability)





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 S lock on A

T1 acquires X lock on A

T1 acquires S lock on B

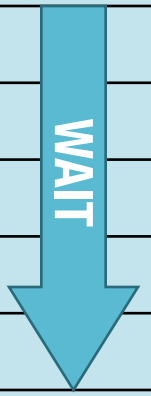
T1 acquires X lock on B

T1 releases all locks

T2 acquires S lock on A

…

T1: Transfer $100 from A to B	T2: Add 10% interest to A and B
Begin	
	Begin
R(A); /A -=100	
W(A)	
R(B); B+=100	
W(B)	
Commit	
	R(A) /A *= 1.1
	W(A)
	R(B), /B *= 1.1
	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 X locks until end of transaction;

Quiz

- 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 trxs holding a lock on this object
  - Mode: shared (read) or exclusive (write) lock
  - List: A wait queue of other trxs requesting a lock on this object

On lock requests for object OID:

- If S lock requested:
  - If its queue is empty and currently not in X mode:
    - Grant the lock, set mode to S and increment the count
- If X lock requested
  - If no one is currently holding the lock (=> queue will be empty too)
    - Grant the lock, set mode to X, set count to 1
- Otherwise, add this trx 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 S mode)
- Note: Lock request handled atomically! (via mutex in OS)

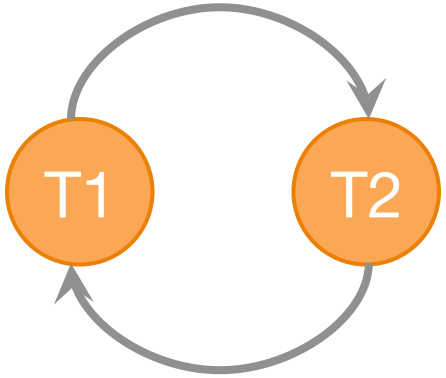
What About Deadlocks?

T1 gets S, then X lock on A

T2 gets S, then X lock on B

T1 waits for lock on B

T2 waits for lock on A



"Waits-for" Graph

T1: Transfer $100 from A to B	T2: Add 10% interest to A and 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
- Approach #1: Lock Mgr maintains waits-for graph
  - Periodically check graph for cycles
  - Abort some Xact to break the cycle
- Approach #2 (Simpler hack): use *time-outs*
  - Abort if no progress made for a while

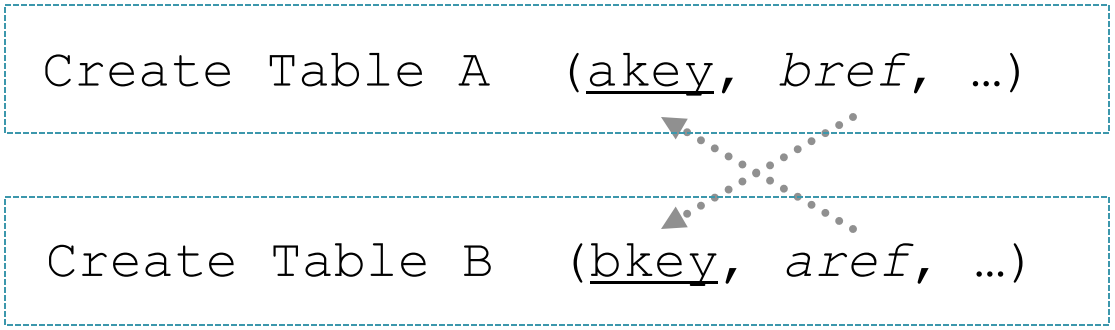
Deadlock Prevention

- Assign priorities to transactions
  - Use timestamp to assign priorities
- Ti requests a lock, held by Tj (in a conflicting mode)
  - Approach #1 (Wait-Die): If Ti has higher priority, it waits; else Ti aborts
  - Approach #2 (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

Transaction and Constraints



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

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

Optional Exercises

- From the textbook: 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?