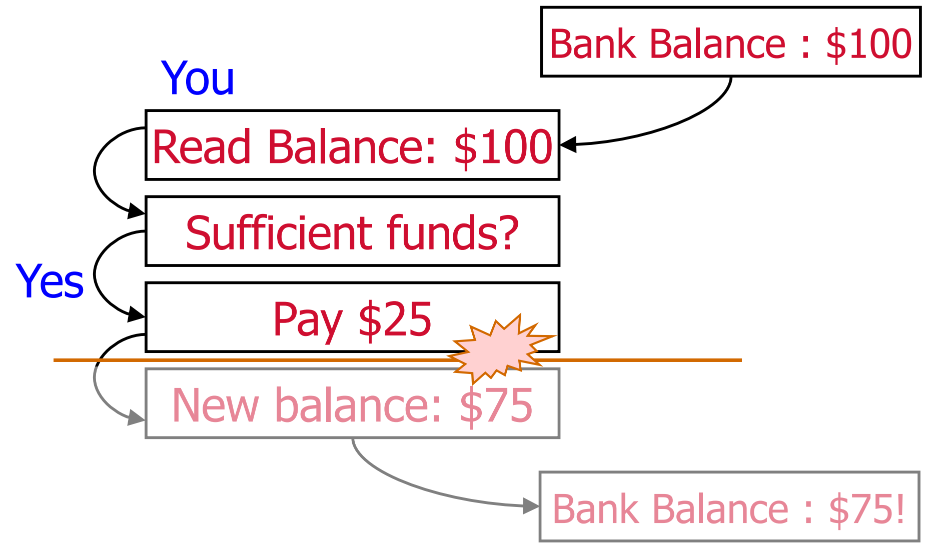
Spring 2017

TRANSACTION MANAGEMENT
[CH 16]

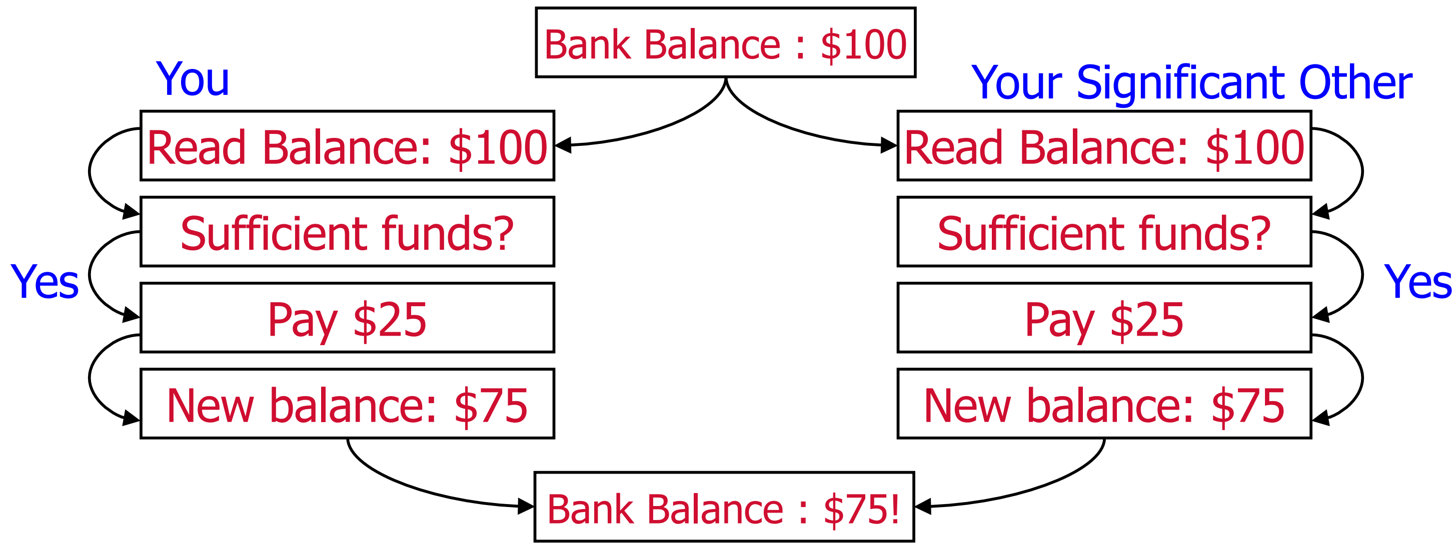
Transaction Management

```
Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);
```



Transaction Management

Read (A);
Check (A > $25);
Pay ($25);
A = A – 25;
Write (A);



- Inconsistency
  - Interleaving actions of different user programs
  - System crash/user abort/…
- Provide the users an illusion of a single-user system
  - Could insist on admitting only one query into the system at any time
    - lower utilization: CPU/IO overlap
    - long running queries starve other queries

What is a Transaction?

- Collection of operations that form a single logical unit
  - A sequence of many actions considered to be one atomic unit of work
- Logical unit:
  - begin transaction …. (SQL) end transaction
- Operations:
  - Read (X), Write (X): Assume R/W on tuples (can be relaxed)
  - Special actions: begin, commit, abort
- Desirable Property: Must leave the DB in a consistent state
  - (DB is consistent when the transaction begins)
  - Consistency: DBMS only enforces ICs specified by the user
  - DBMS does not understand any other semantics of the data

The ACID Properties

TM
Xact. Mgmt.
(logging)

- **A**tomicity: All actions in the Xact happen, or none happen.

User

- **C**onsistency: Consistent DB + consistent Xact ⇒ consistent DB

CC
Concurrency Ctrl.
(locking)

- **I**solation: Execution of one Xact is isolated from that of other Xacts.

RM
Recovery Mgmt.
(WAL, …)

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

```
Begin
   Read (A);
   A = A – 25;
   Write (A);
   Read (B);
   B = B + 25;
   Write (B);
Commit
```

Schedules

- <u>Schedule</u>: An interleaving of actions from a set of Xacts, where the actions of any one Xact are in the original order.
  - Actions of Xacts *as seen by the DB*
  - *Complete schedule* : each Xact ends in commit or abort
  - *Serial schedule* : No interleaving of actions from different Xacts.
- Initial State + Schedule $\rightarrow$ Final State

Time

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

Acceptable Schedules

- One sensible "isolated, consistent" schedule:
  - Run Xacts one at a time (serial schedule)
- <u>Serializable</u> schedules:
  - Final state is what ***some complete*** serial schedule of ***committed*** transactions would have produced.
  - Can different serial schedules have different final states?
    - Yes, all are "OK"!
  - Aborted Xacts?
    - ignore them for a little while (made to 'disappear' using logging)
  - 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

Serializability Violations

- @Start (A,B) = (1000, 100)
  - End (990, 210)
- T1→T2:
  - (900, 200) → (990, 220)
- T2→T1:
  - (1100, 110) → (1000, 210)
- W-R conflict: Dirty read
  - *Could* lead to a non-serializable execution
- Also R-W and W-W conflicts

T1: Transfer $100 from A to B	*T2: Add 10% interest to A & 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	

Database Inconsistent

More Conflicts

- RW Conflicts (Unrepeatable Read)
  - $R_{T2}(X) \rightarrow W_{T1}(X)$, T1 overwrites what T2 read.
  - $R_{T2}(X) \rightarrow W_{T1}(X) \rightarrow R_{T2}(X)$. T2 sees a different X value!
- WW Conflicts (Overwriting Uncommited Data)
  - T2 overwrites what T1 wrote.
    - E.g. : Students in the same group get the same project grade.
    - T_P: W (X=A), W (Y=A) T_{TA}: W (X=B), W (Y=B)
    - $W_P(X{=}A) \rightarrow W_{TA}(X{=}B) \rightarrow W_{TA}(Y{=}B) \rightarrow W_P(Y{=}A)$
      [Note: no reads]
  - Usually occurs in conjunction with other anomalies.
    - Unless you have “blind writes”.

Now, Aborted Transactions

- Serializable schedule: Equivalent to a serial schedule of *committed* Xacts.
  - as if aborted Xacts *never happened.*
- Two Issues:
  - How does one undo the effects of a Xact?
    - We'll cover this in logging/recovery
  - What if another Xact sees these effects??
    - Must undo that Xact as well!

Cascading Aborts

- Abort of T1 requires abort of T2!
  - **Cascading Abort**

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

Cascading Aborts

- Abort of T1 requires abort of T2!
  - **Cascading Abort**
- Consider commit of T2
  - Can we undo T2?
- *Recoverable* schedule: Commit only after all xacts that supply dirty data have committed.

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

Cascading Aborts

- *ACA (avoids cascading abort)* schedule
  - Transaction only reads committed data
  - One in which cascading abort cannot arise.
  - Schedule is also recoverable

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

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

Locking: A Technique for C. C.

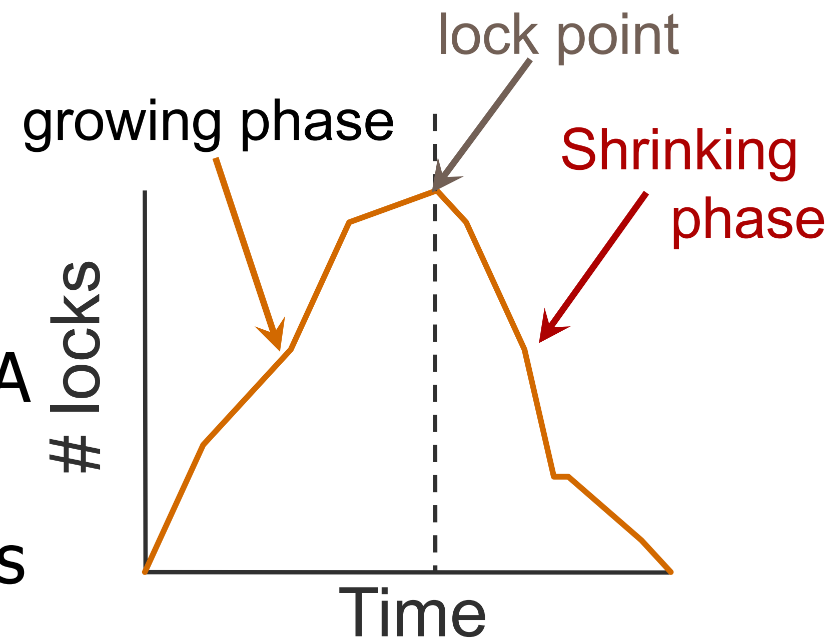
- Concurrency control usually done via locking.
- Lock info maintained by a "lock manager":
  - Stores (XID, RID, Mode) triples.
    - This is a simplistic view; suffices for now.
  - Mode $\in$ {S,X}
  - Lock compatibility table:

	--	S	X
--	√	√	√
S	√	√	
X	√		

- If a Xact can't get a lock
  - Suspended on a wait queue
- When are locks acquired?
  - Buffer manager call!

Two-Phase Locking (2PL)

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



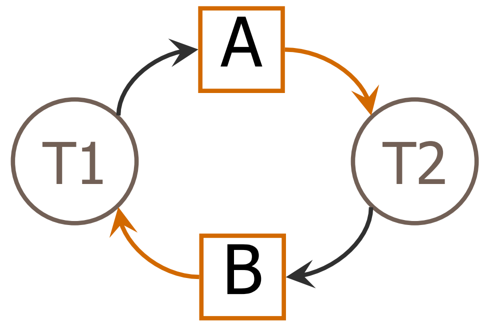
- Strict 2PL:
  - Hold all locks until end of Xact
  - Guarantees serializability, and ACA too!
    - Note ACA schedules are always recoverable

Schedule with Locks

T1: Transfer $100 from A to B	*T2: Add 10% interest to A & 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	

T1	*T2*
begin	
	begin
X(A)	
R(A)	
W(A)	
	X(A) – Wait!
X(B)	
R(B)	
W(B)	
U_x(A), U_x(B)/commit	
	R(A)
	W(A)
	...

Deadlocks

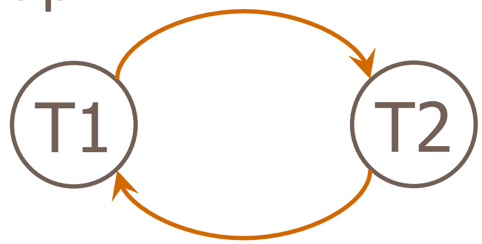


$$\boxed{X_{T1}(B),\ X_{T2}(A),\ S_{T1}(A),\ S_{T2}(B)}$$

- Deadlocks can cause the system to wait forever.
- Need to detect deadlock and break, or prevent deadlocks
- Simple mechanism: timeout and abort
- More sophisticated methods exist

Precedence Graph

- Precedence (or Serializability) graph:
  - Nodes = Committed Xacts
  - Conflicts = Arcs

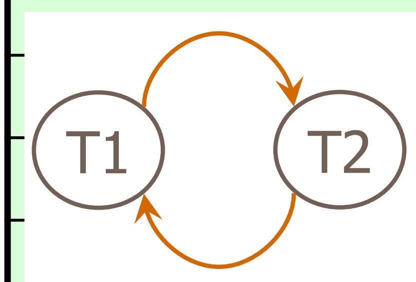


- *Conflict equivalent*:
  - Same sets of actions
  - Conflicting actions in the same order
- *Conflict serializable*: Conflict equivalent to a serial schedule

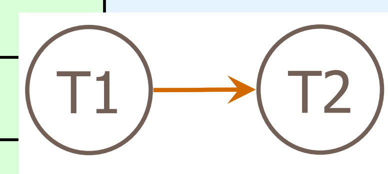
T1: Xfer. $100 from A to B	*T2: Add 10% interest*
begin	
	begin
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	commit
R(B)	
W(B)	
commit	

Schedule with Locks

T1: Transfer $100 from A to B	*T2: Add 10% interest to A & 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	



T1	*T2*
begin	
	begin
X(A)	
R(A)	
W(A)	
	X(A) – Wait!
X(B)	
R(B)	
W(B)	
$U_x(A)$, $U_x(B)$/commit	
	R(A)
	W(A)
	…



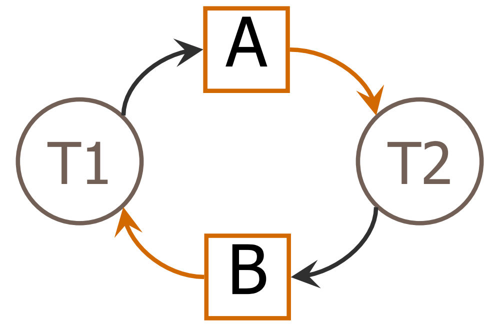
Conflict Serializability & Graphs

Theorem: A schedule is conflict serializable iff its precedence graph is acyclic

Theorem: 2PL ensures that the precedence graph will be acyclic

- Why Strict 2PL?
  - Guarantees ACA
    - read only committed values
  - How? Write locks until EOT
    - No WW or WR => on abort replace original value

Deadlocks



$$\boxed{X_{T1}(B),\ X_{T2}(A),\ S_{T1}(A),\ S_{T2}(B)}$$

- Deadlocks can cause the system to wait forever.
- Need to detect deadlock and break, or prevent deadlocks
- Detect deadlock
  – Draw a lock graph. Cycles implies a deadlock
- Alternative ways of dealing with deadlock
  – Break Deadlock
    - On each lock request “update the lock graph”. If a cycle is detected, abort one of the transactions. The aborted transaction is restarted after waiting for a time-out interval.
  – Prevent deadlock
    - Assign priorities to the transactions. If a transaction, T1, requests a lock that is being held by another transaction, T2, with a lower priority, then T1 “snatches” the lock from T2 by aborting T2 (which frees up the lock on the resource). T2 is then restarted again after a time-out.

(not in the official course syllabus)

Transaction Support in SQL

- Transaction boundary
  - Begin implicitly, or end by *Commit work, Rollback work*
  - For long running transactions: *Savepoint*
- Transaction characteristics
  - Diagnostic size: # error messages…
  - Access mode: Read only, Read Write
  - Isolation level
    - Serializable: default (long-term R/W locks on phantoms too)
    - Repeatable reads: (long-term R/W locks on real objects)
      - Read only committed records
      - Between two reads by the same Xact, no updates by another Xact
    - Read committed (long-term W locks/short-term R locks)
      - Read only committed records
    - Read uncommitted (Read only, no R locks!)

(not in the official course syllabus)

Phantom Problem

- T1: Scan Sailors for the oldest sailor for ratings 1 and 2
  - Assume that at the start the oldest sailor with rating 1 has age 80, oldest sailor with rating 2 has age 90, and the second oldest sailor with rating 2 is 85 years old
  - T1 identifies pages with sailors having a rating 1, and locks these pages. It computes the first tuple (rating = 1, oldest-age = 80)
  - T1 then gets ready to lock pages with sailor tuples with rating 2. However, before it can get started, T2 arrives
- T2: Inserts a tuple with rating 1 and age 99, and deletes the oldest sailor with rating 2 (whose age is 90)
  - The new tuple is inserted into a page that doesn't have a sailor with rating 1 or 2, and is not locked by T1
  - T2 commits
- T1 now resumes and completes looking at sailors with rating 2.
- The final answer produced by T1 is (1,80) (2,85) does not correspond to either of the two serial schedules:
  - T1 -> T2 Answer: (1, 80), (2, 90)
  - T2 -> T1 Answer: (1, 99), (2, 85)

(not in the official course syllabus)

Transaction and Constraints

Create Table A (akey, *bref*, …)

Create Table B (bkey, *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
  - DEFERRED: Check at commit time.
  - IMMEDIATE: Check immediately

The ACID Properties

TM
Xact. Mgmt.
(logging)

- Atomicity: All actions in the Xact happen, or none happen.

User

- Consistency: Consistent DB + consistent Xact ⇒ consistent DB

CC
Concurrency Ctrl.
(locking)

- Isolation: Execution of one Xact is isolated from that of other Xacts.

RM
Recovery Mgmt.
(WAL, …)

- Durability: If a Xact commits, its effects persist.