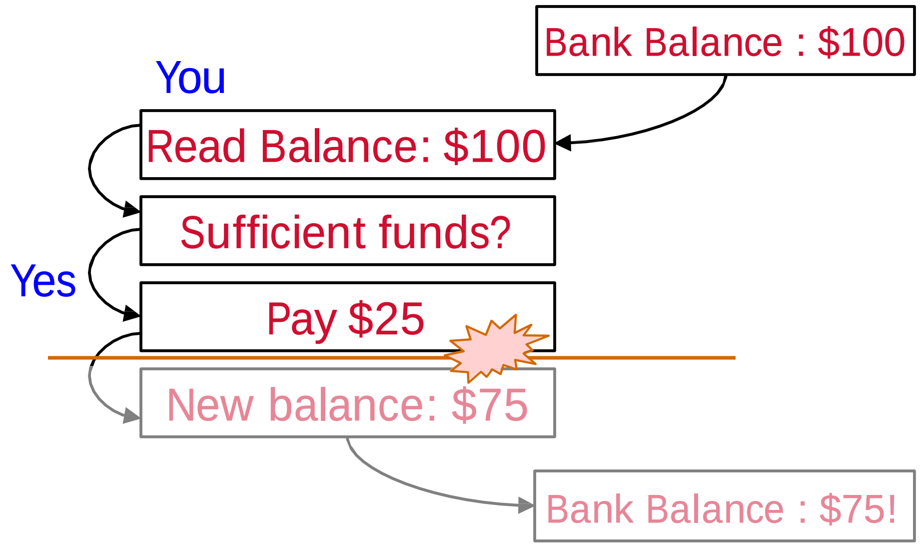
TRANSACTION MANAGEMENT

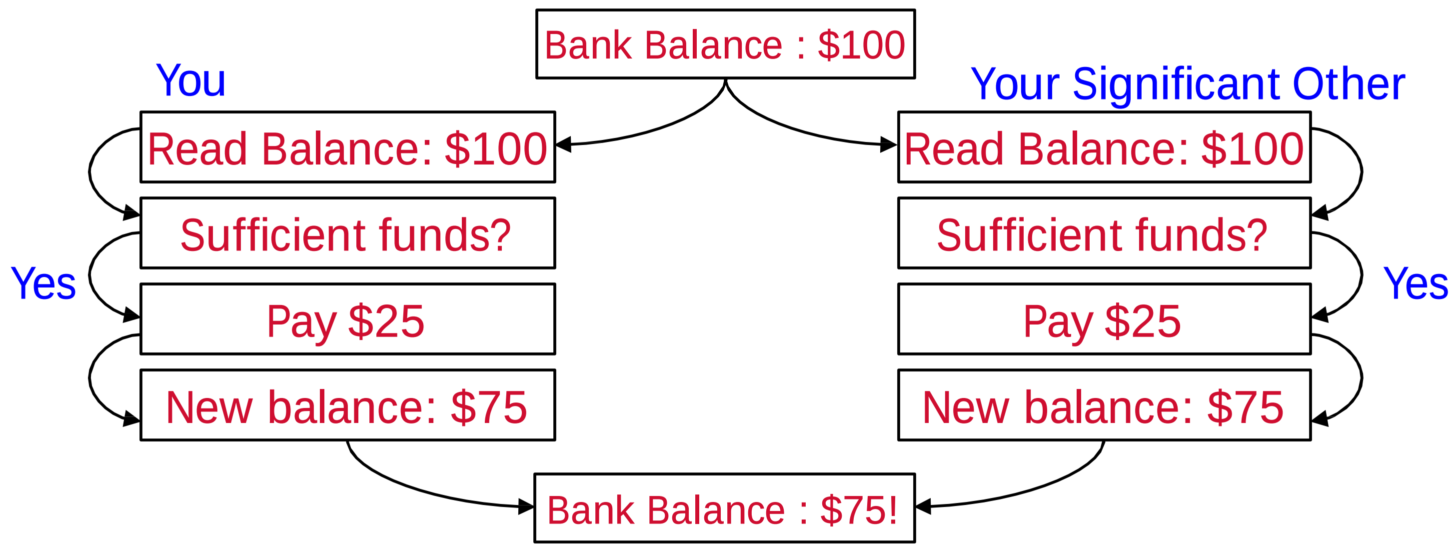
Transaction Management

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 integrity constraints (IC) 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.

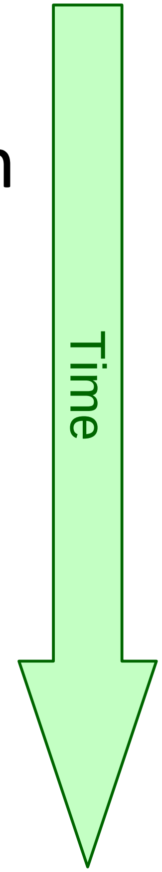
RM
Recovery Mgmt. (WAL, …)

- **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



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)
- Serializable 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.

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?
    - 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.

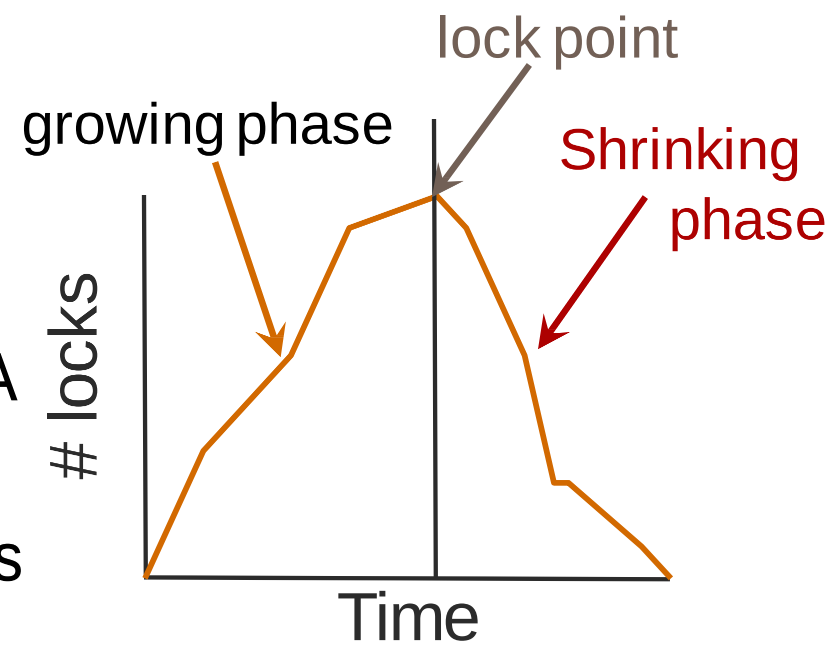
- Concurrency control usually done via locking.
- Lock info maintained by a "lock manager":
  - Stores (XID, RID, Mode) triples.
    - This is a simplistic view; suffices for now.
  - Mode $\in$ {S,X}
  - Lock compatibility table:

	--	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

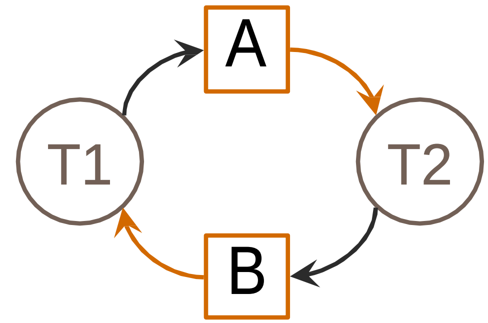
Schedule with Locks

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

Conflict Serializability & Graphs

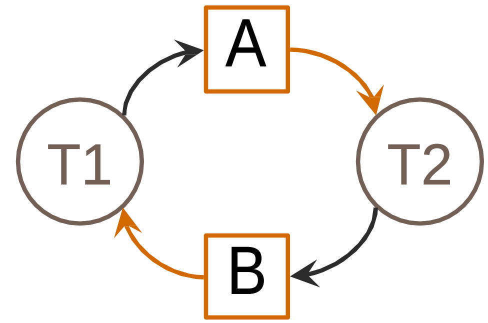
Theorem: A schedule is conflict serializable iff its precedence graph is acyclic

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

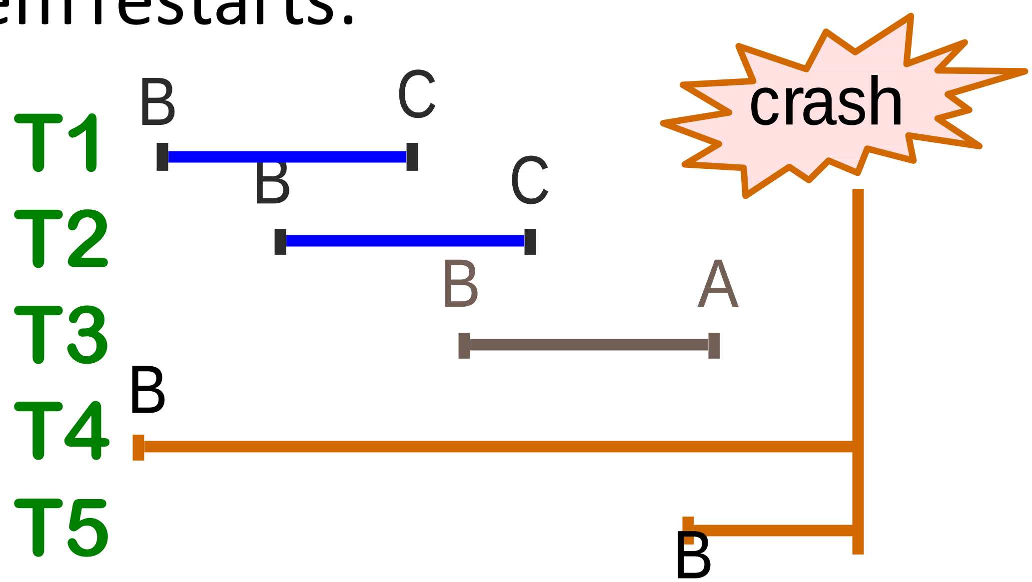
$$\boxed{X_{T1}(B),\ X_{T2}(A),\ S_{T1}(A),\ S_{T2}(B)}$$



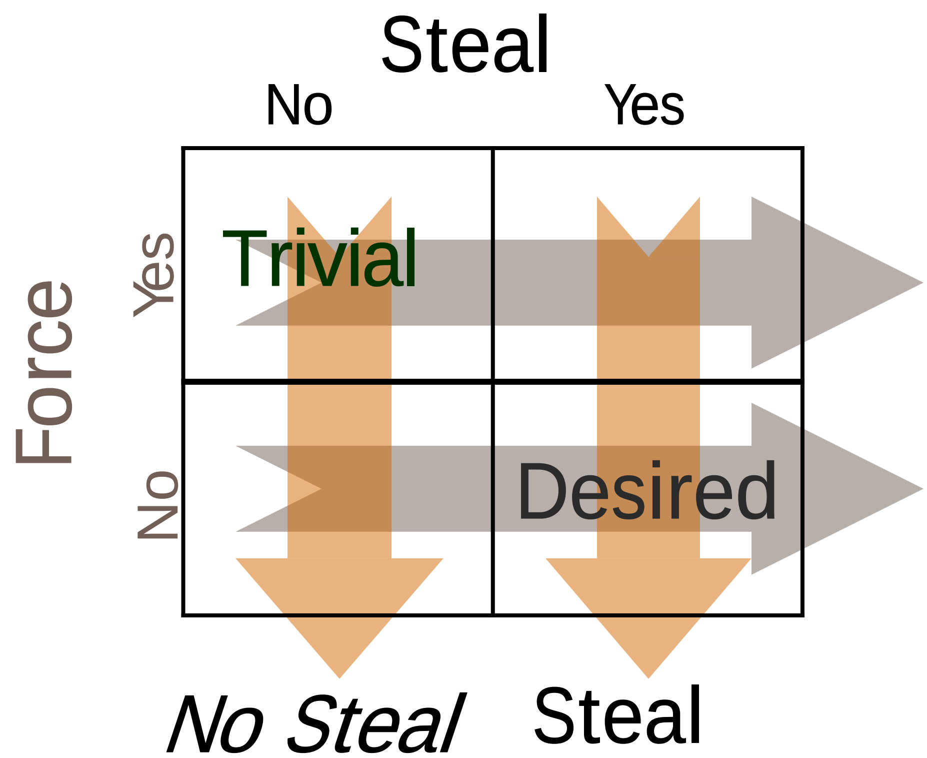
- Deadlocks can cause the system to wait forever.
- Need to detect deadlock and break, or prevent deadlocks
- Detect deadlock
  - Draw a lock graph. Cycles implies a deadlock
- Alternative ways of dealing with deadlock
  - Break Deadlock
    - On each lock request "update the lock graph". If a cycle 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.

Ensuring Atomicity & Durability

- Atomicity:
  - Transactions may abort ("Rollback") -> no effects should be seen
- Durability:
  - What if DBMS stops running? (Causes?)
- Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable.
  - T4 & T5 should be aborted



Buffer Pool: Sharing & Writing



Poor throughput, but works

Force

- *Poor response time, but durable*

No Force

- Crash before a page is flushed to disk
- Soln: Force a short summary @ commit (logs). Use this to REDO

Steal

- Page being stolen (and flushed) was modified by an uncommitted Xact T
- If T aborts, how is atomicity enforced?
- Soln: Remember old value (logs). Use this to UNDO

Basic Idea: Logging

- Record information, for every change, in a *log*.
  - Sequential writes to log (put it on a separate disk).
  - Stored in stable storage to survive system crash
    - disk mirroring
  - Each record has a log sequence number (LSN)
  - Log record contains:
    - <prevLSN, XID, type, … >
    - and additional control info (which we'll see soon)
    - Note: the log records for a transaction are chained by prevLSN

Write-Ahead Logging (WAL)

- The Write-Ahead Logging Protocol:
  1. Must force the log record for an update *before* the corresponding data page gets to storage.
  2. Must write all log records for a Xact *before commit*.
- #1 guarantees Atomicity.
- #2 guarantees Durability.

If DB says TX **commits**, TX effect **remains** after database crash

DB can **undo actions** and help us with **atomicity**

Normal Execution of a Xact

- Series of reads & writes, followed by commit or abort.
  - Updates are "in place": i.e., data on disk is overwritten
  - We will assume that write is atomic on disk.
    - In practice, additional details to deal with non-atomic writes.
- Strict 2PL.
- STEAL, NO-FORCE buffer management, with Write-Ahead Logging.

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 $\Rightarrow$ 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.

Postgres

- PostgreSQL transactions follow ACID compliance
- Atomicity & Durability: WAL logging, Continuous Archiving and Point-in-Time Recovery (PITR)
- Consistency: Multiversion Concurrency Control, MVCC
- Isolation: *Serializable Snapshot* Isolation (SSI)

- Every command we've executed in psql has been implicitly wrapped in a transaction

- Explicit transaction:

```
BEGIN TRANSACTION;
  DELETE FROM events;
ROLLBACK;
SELECT * FROM events;
```

END; or **COMMIT**; Commits the current transaction

Install MongoDB

We will be using the free MongoDB Community Edition Database server.

https://www.mongodb.com/try/download/community
Download a build for your OS

After download completes, follow the Installation instructions
https://docs.mongodb.com/manual/installation/

From a terminal window in your computer, run the mongoDB server (`mongod`).

Then run the mongo shell client (`mongo`) -
https://docs.mongodb.com/manual/mongo/

Take a screenshot of your terminal window and upload to ICON in the dropbox.

To exit the shell, type **quit()** or use the **<Ctrl-C>** shortcut