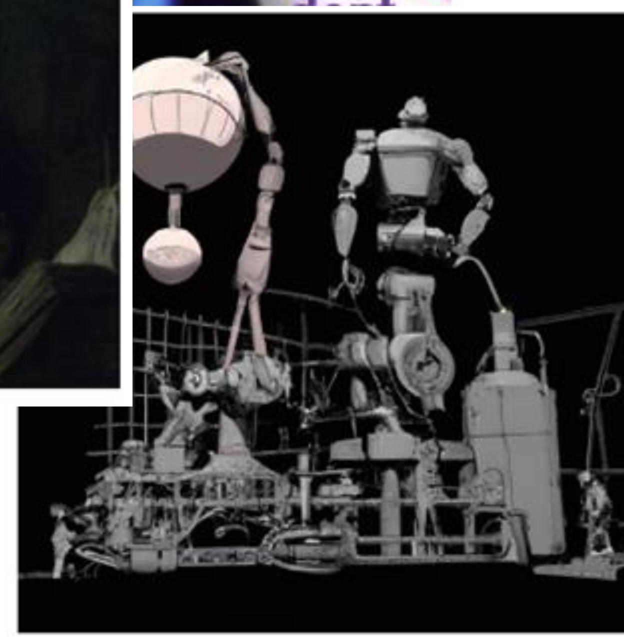
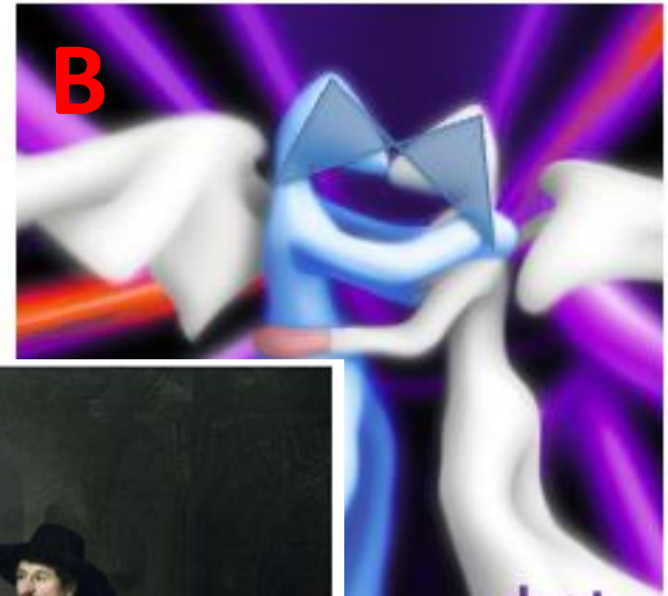
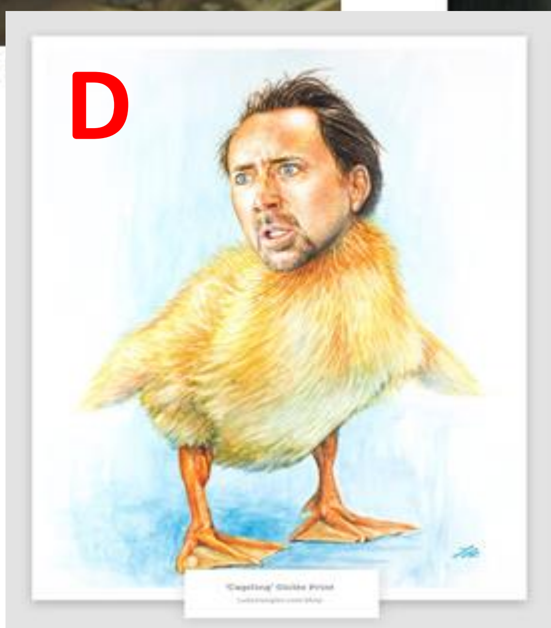
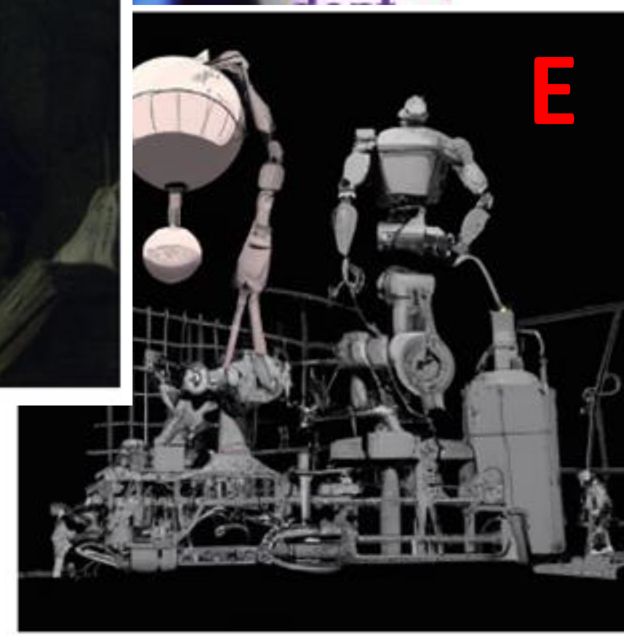
The Lecture Art Collection So far



https://clicker.mit.edu/6.5830/







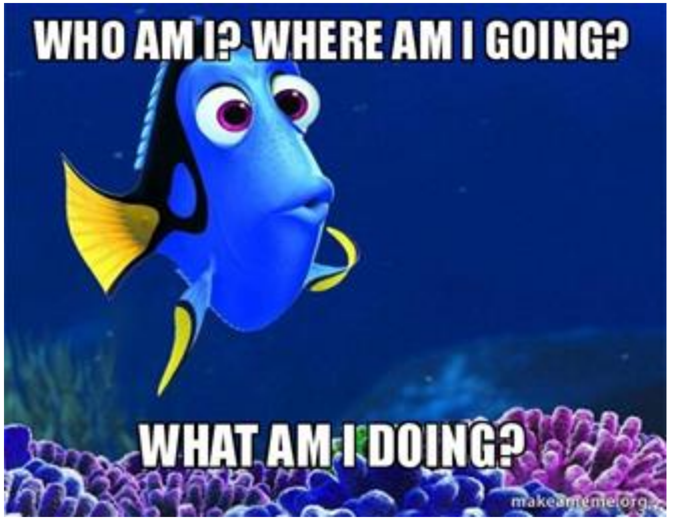
Lecture 10
Transactions

Project meetings:
to be scheduled soon!

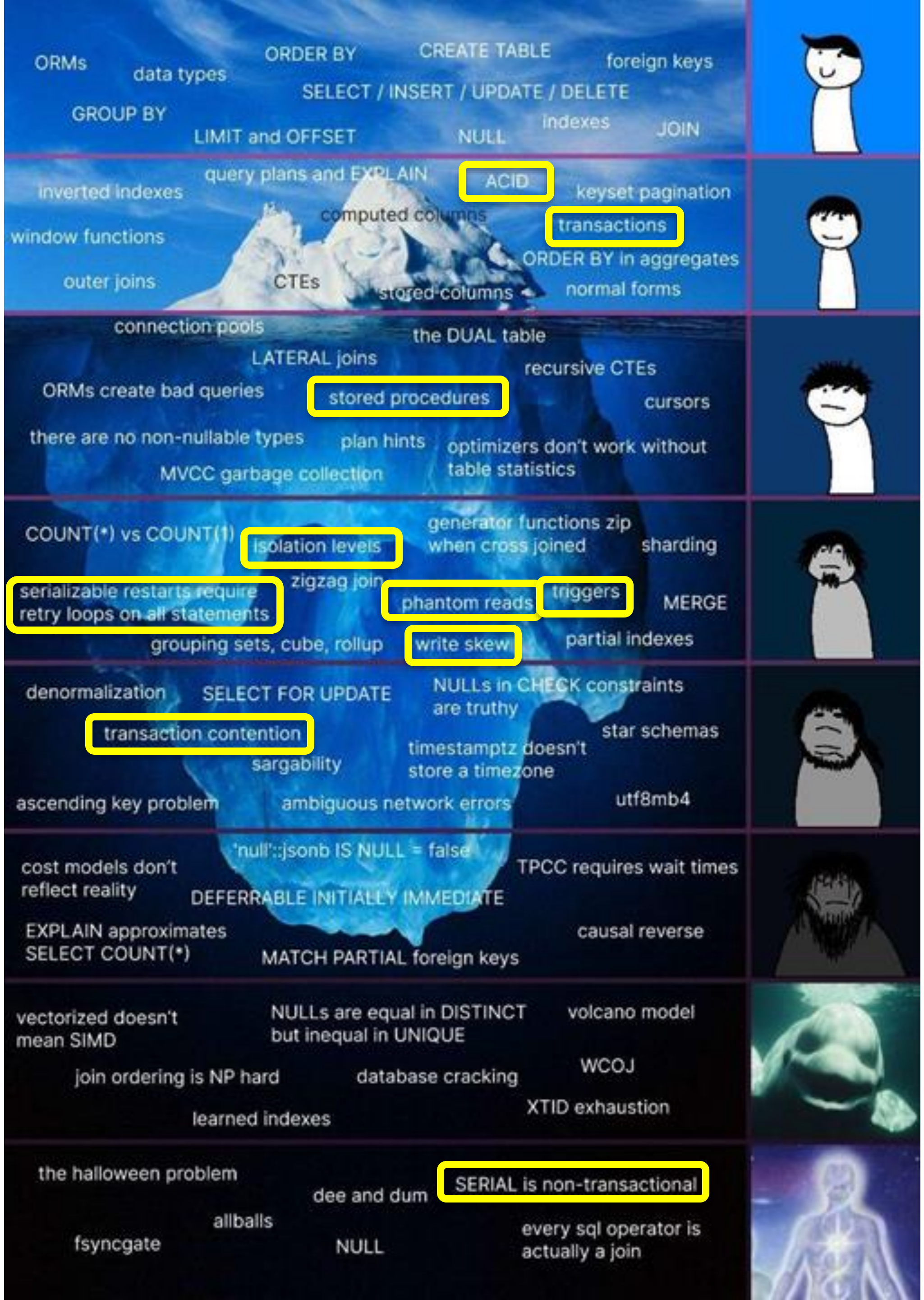
“The Moneylender and His Wife,” Quentin Matsys, 1514

10/16/2024

Where Are We?



- So far:
  - Studied relational model & SQL
  - Learned basic architecture of a database system
  - Studied different operator implementations
  - Looked at several data layouts
  - Saw how query optimizer works with statistics to select plans and operators
- What next:
  - *Concurrency Control and Recovery*: How to ensure correctness in the presence of modifications and failures to the database
  - Distributed and parallel query processing
  - "Advanced Topics"



Next 4 lectures

Transactions

- Group related sequence of actions so they are "all or nothing"
  - If the system crashes, partial effects are not seen
  - Other transactions do not see partial effects

- A set of implementation techniques that provides this abstraction with good performance

ACID Properties of Transactions

- **A** tomicity – many actions look like one; “all or nothing”
- **C** onsistency – database preserves invariants
- **I** solation – concurrent actions don’t see each other’s results
- **D** urability – completed actions in effect after crash (“recoverable”)

Concurrent Programming Is Hard

A = ~~0~~ ~~1~~ 1

- Example:

```
T1
t = A
t = t + 1
A = t
```

```
T2
t = A
t = t + 1
A = t
```

- Looks correct!
- But maybe not if other updates to A are interleaved!
- Suppose T1 increment runs just before T2 increment
  - T1 increment will be lost
- Conventional approach: programmer adds locks
  - But must reason about other concurrent programs

Transactions Dramatically Simplify Concurrent Programming

- Guarantees that concurrent actions are *serially equivalent*
  - I.e., appear to have run one after the other
- Programmer does not have to think about what is running at the same time!
- **One of the big ideas in computer science**

SQL Syntax

- BEGIN TRANSACTION
  - Followed by SQL operations that modify database
- COMMIT: make the effects of the transaction durable
  - After COMMIT returns database guarantees results present even after crash
  - And results are visible to other transactions
- ABORT: undo all effects of the transaction

This Lecture: Atomicity

- **A**tomicity – many actions look like one; “all or nothing”
- In reality, actions take time!
  - To get atomicity, to prevent multiple actions from interfering with each other
  - I.e., are **I**solated
- Will return to **D**urability in 2 lectures
  - E.g., how to *recover* a database after a crash into a state where no partial transactions are present

Consistency

- Preservation of invariants
- Usually expressed in terms of constraints
  - E.g., primary keys / foreign keys
  - Triggers
- Example: no employee makes more than their manager
- Requires ugly non-SQL syntax (e.g. PL/pgSQL)
- Often done in the application

Postgres PL/pgSQL Trigger Example

```
CREATE FUNCTION sal_check() RETURNS trigger AS $sal_check$
    DECLARE
        mgr_sal integer;
    BEGIN
        IF NEW.salary IS NOT NULL THEN
            SELECT INTO mgr_sal salary
                FROM emp
                JOIN manages
                    ON NEW.eid = manages.eid
                    AND emp.eid = manages.eid
                LIMIT 1;
            IF (mgr_sal < NEW.salary) THEN
                RAISE EXCEPTION 'employee cannot make more than manager';
            END IF;
        END IF;
        RETURN NEW;
    END;
$sal_check$ LANGUAGE plpgsql;
CREATE TRIGGER eid_sal BEFORE INSERT OR UPDATE ON emp
FOR EACH ROW EXECUTE FUNCTION sal_check();
```

NEW is the record being added

mgr_sal is a local variable

Query finds the salary of one manager

Check salary (if no manager, mgr_sal is NULL)

Declare that we want to call sal_check every time a record changes or is added to emp

How Can We Isolate Actions?

- Serialize execution: one transaction at a time
- Problems with this?
  - No ability to use multiple processors
  - Long running transactions *starve* others

- Goal: allow *concurrent* execution while preserving *serial equivalence*

- *Concurrency control* algorithms do this

Serializability

- An ordering of actions in concurrent transactions that is serially equivalent

T1	T2
RA	
WA	
	RA
	WA
RB	
WB	
	RB
	WB

RA: Read A
WA: Write A, may depend on anything read previously

A/B are “objects” – e.g., records, disk pages, etc

Assume arbitrary application logic between reads and writes

Serially equivalent to T1 then T2

Serializability

- An ordering of actions in concurrent transactions that is serially equivalent

T1	T2
RA	
	RA
	WA
WA	
RB	
WB	
	RB
	WB

RA: Read A

WA: Write A, may depend on anything read previously

A/B are "objects" – e.g., records, disk pages, etc

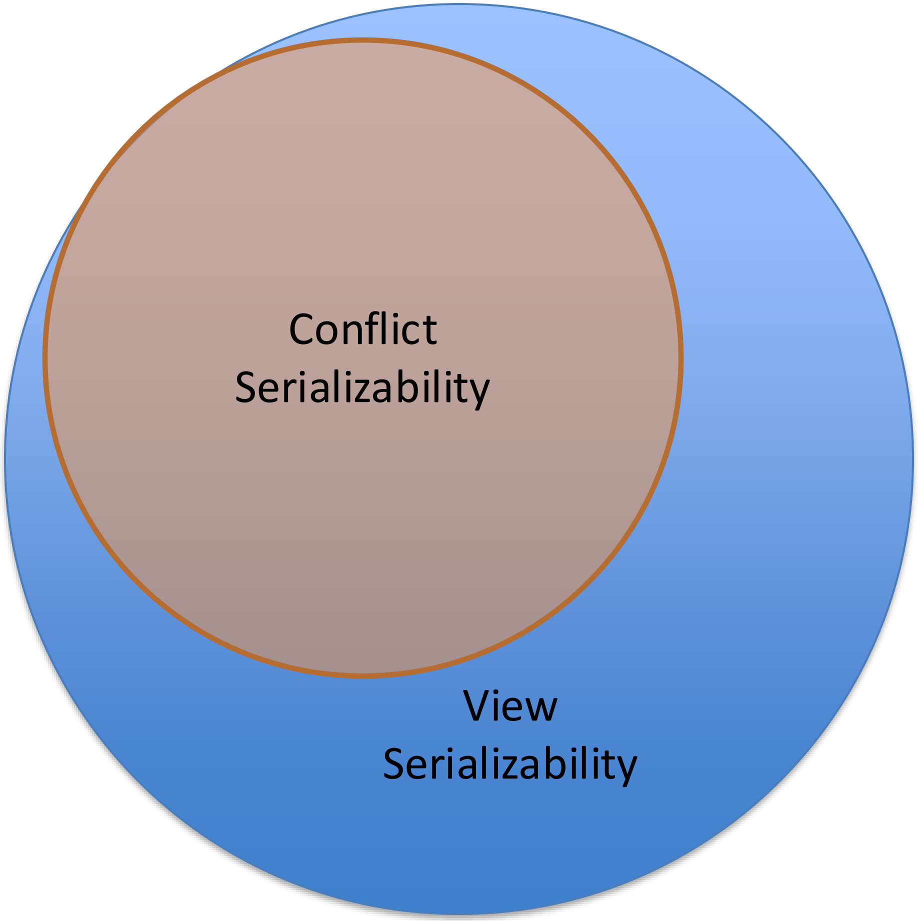
Assume arbitrary application logic between reads and writes

Not serially equivalent – T2's write to A is lost, couldn't occur in a serial schedule

In T1-T2, T2 should see T1's write to A

In T2-T1, T1 should see T2's write to A

Testing for Serializability



*Any schedule that is **conflict serializable** is **view serializable**, but not vice-versa*

View Serializability

A particular ordering of instructions in a schedule S is *view equivalent* to a serial ordering S' iff:

- Every value read in S is the same value that was read by the same read in S'.
- The final write of every object is done by the same transaction T in S and S’
- Less formally, all transactions in S “view” the same values they view in S', and the final state after the transactions run is the same.

View Serializability Example

S

T1	T2
RA=A1	
WA→A2	
	RA = A2
	WA→A3
RB=B1	
WB→B2	
	RB=B2
	WB→B3

Same valu
read in bo
schedules

Every value read in S is the same value that was read by the same read in S'.

The final write of every object is done by the same transaction T in S and S'

https://clicker.mit.edu/6.5830/

Is the following schedule view serializable?

T1	T2
RA=A1	
	RA=A1
	WA->A2
	WB->B2
WB->B3	

A) Yes

B) No

C) I don’t know

A particular ordering of instructions in a schedule S is *view equivalent* to a serial ordering S' iff:

- Every value read in S is the same value that was read by the same read in S'.
- The final write of every object is done by the same transaction T in S and S’

View Serializability Limitations

- Must test against each possible serial schedule to determine serial equivalence
  - NP-Hard!

(For N concurrent transactions, there are 2^N possible serial schedules)

- No protocol to ensure view serializability as transactions run

- *Conflict serializability* addresses both points

Conflicting Operations

- Two operations are said to conflict if:
  - Both operations are on the same object
  - At least one operation is a write
  - E.g.,
    - $T1_{WA}$ conflicts with $T2_{RA}$, but
    - $T1_{RA}$ does not conflict with $T2_{RA}$

T2 \ T1	R	W
R	✓	✗
W	✗	✗

Conflict Serializability

A schedule is *conflict serializable* if it is possible to swap non-conflicting operations to derive a serial schedule.

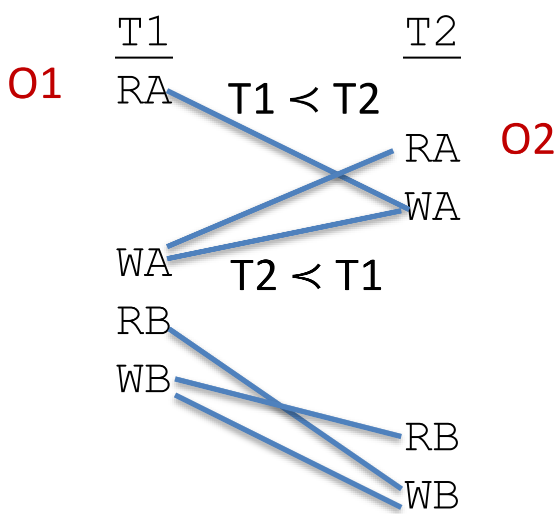
Equivalently

For all pairs of conflicting operations {O1 in T1, O2 in T2} either

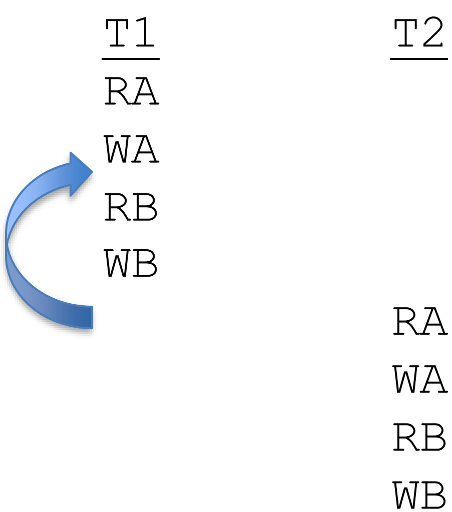
- O1 always precedes O2, or
- O2 always precedes O1.

T1 $\prec$ T2 : “T1 precedes T2”

Example



Not conflict serializable!



In conflict serializable schedule, can reorder non-conflicting ops to get serial schedule

For all pairs of conflicting operations {O1 in T1, O2 in T2} either
O1 always precedes O2, or
O2 always precedes O1.

Precedence Graph

Given transactions Ti and Tj,
Create an edge from Ti→Tj if:

- Ti reads/writes some A before Tj writes A

  $RA_{Ti} \prec WA_{Tj}$ or $WA_{Ti} \prec WA_{Tj}$

  or

- Ti writes some A before Tj reads A

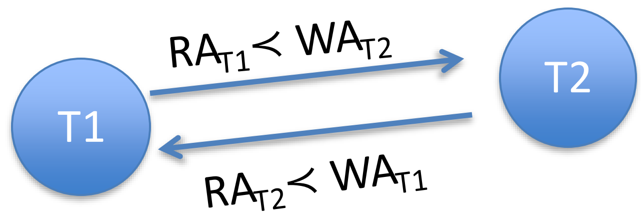
  $WA_{Ti} \prec RA_{Tj}$

If there are cycles in this graph, schedule is not conflict serializable

Non-Serializable Example

T1	T2
RA	
	RA
	WA
WA	
RB	
WB	
	RB
	WB

Precedence Graph



Cycle!

Create an edge from Ti→Tj if:

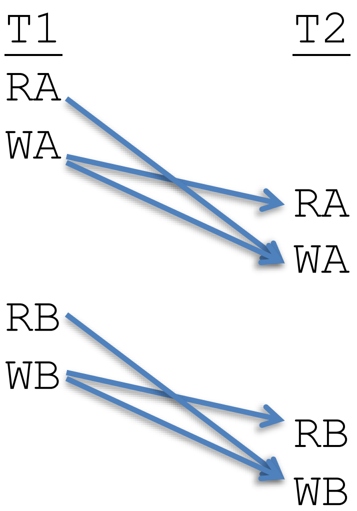
Ti reads/writes some A before Tj writes A, or

$RA_{Ti} \prec WA_{Tj}$ or $WA_{Ti} \prec WA_{Tj}$

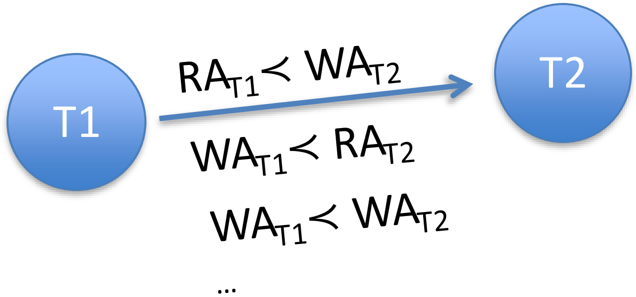
Ti writes some A before Tj reads A

$WA_{Ti} \prec RA_{Tj}$

Serializable Example



Precedence Graph



No Cycles!

Create an edge from Ti→Tj if:

Ti reads/writes some A before Tj writes A, or

$RA_{Ti} \prec WA_{Tj}$ or $WA_{Ti} \prec WA_{Tj}$

Ti writes some A before Tj reads A

$WA_{Ti} \prec RA_{Tj}$

Recap: 3 Ways to Test for Conflict Serializabiliy

1. Check: For all pairs of conflicting operations {O1 in T1, O2 in T2} either
   1. O1 always precedes O2, or
   2. O2 always precedes O1.
2. Swap non-conflicting operations to get serial schedule
3. Build precedence graph, check for cycles

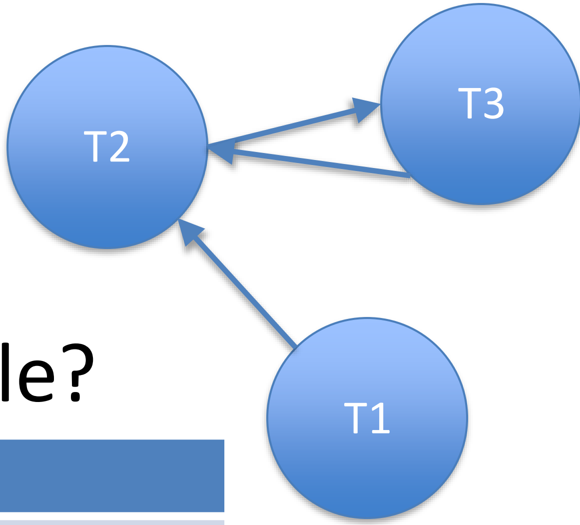
Clicker: https://clicker.mit.edu/6.5830/

- Is this schedule conflict serializable?

A) Yes
B) No
C) ???

T1	T2	T3
RA		
	RB	
WA		
		RB
	WB	
		WB
	RA	
	WA	
COMMIT	COMMIT	COMMIT

Clicker



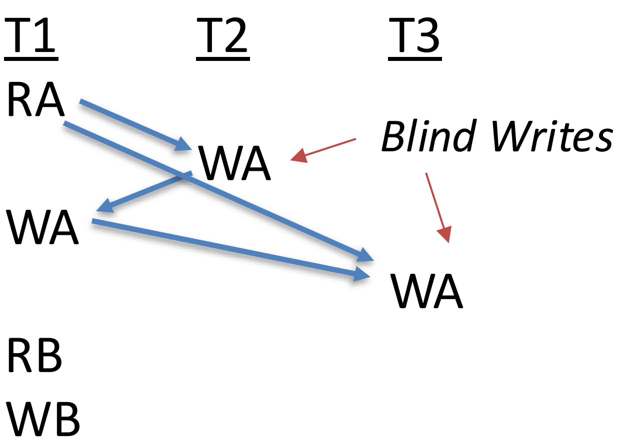
- Is this schedule conflict serializable?

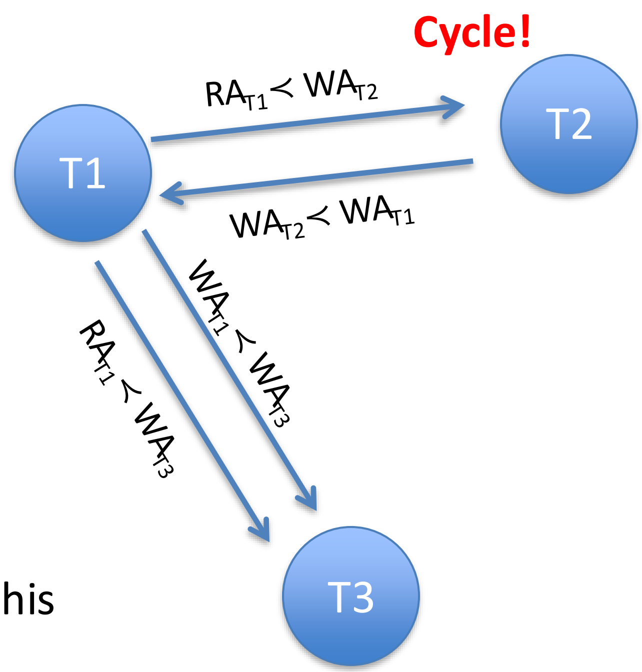
T1	T2	T3
RA		
	RB	
WA		
		RB
	WB	
		WB
	RA	
	WA	
COMMIT	COMMIT	COMMIT

No!

View vs Conflict Serializable

- Testing for view serializability is NP-Hard
  - Have to consider all possible orderings
- Conflict serializability used in practice
  - Not because of NP-Hardness
  - Because we have a way to enforce it as transactions run
- Example of schedule that is view serializable but not conflict serializable:





Equivalent to T1, T2, T3
Conflict serializability does not permit this
Only happens with *blind writes*

Implementing Conflict Serializability

- Several different protocols
- Today: Two Phase Locking (2PL)
- Basic idea:
  - Acquire a shared (S) lock before each read of an object
  - Acquire an exclusive (X) lock before each write of an object
- Several transactions can hold an S lock
- Only one transaction can hold an X lock
- If a transaction cannot acquire a lock it waits ("blocks")

T2 \ T1	R	W
R	✓	✗
W	✗	✗

T2 \ T1	S	X
S	✓	✗
X	✗	✗

Lock Compatibility Table

Conflicting operations (from def. of conflict serializability) are not compatible with each other

When to Release Locks

- After each op completes?
- Or after xaction is done with variable?
- No! Example of problem →
- T2 “sneaks in” and updates A and B before T1 updates B

T1	T2
Xlock A	
RA	
WA	
Rel A	
	Xlock A
	RA
	WA
	Xlock B
	RB
	WB
	Rel A,B
Xlock B	
RB	
WB	
Rel B	

This schedule is not serializable

Solution: Two Phase Locking

- *A transaction cannot release **any** locks until it has acquired **all** of its locks*

- Two-phase locking has a ***growing phase*** and a ***shrinking phase***

Example, Revisited

- Rule: A transaction cannot release any locks until it has acquired all of its locks

	T1	T2
	Xlock A	
	RA	
	WA	
Not allowed →	Rel A	
		Xlock A
		RA
		WA
		Xlock B
		RB
		WB
		Rel A,B
	Xlock B	
	RB	
	WB	
	Rel B	

This schedule is not serializable

Example, Revisited

- Rule: A transaction cannot release any locks until it has acquired all of its locks
- Serial schedule defined by *lock points*
  - Where they acquire last lock

	T1	T2
	Xlock A	
	RA	
	WA	
	Xlock B ← *Lock point*	
Acquired all locks so can release →	Rel A	
		Xlock A
		RA
		WA
	RB	
	WB	
	Rel B	
		Lock point → Xlock B
		RB
		WB
		Rel A,B

*This schedule *is* serializable*

Correctness Intuition

- Once a transaction T reached its lock point…
  - T's place in serial order is set
  - Any transactions that haven't acquired all their locks can't take any conflicting actions until after T releases locks
    - **Ordered later**
  - Any transactions which already have all their locks must have completed their conflicting actions (released their locks) before T can proceed
    - **Ordered earlier**

Two Phase Locking (2PL) Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- Release locks only after last lock has been acquired, and ops on that object are finished

Can you think of any potential problems with 2PL?

Refining 2PL

- Problems:
  - Deadlocks
  - Cascading Aborts

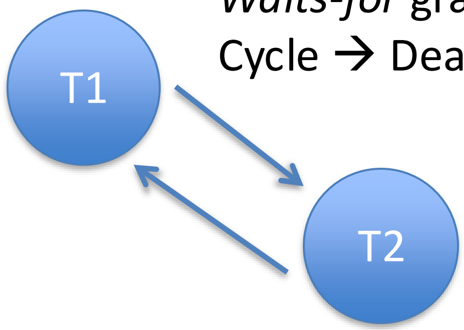
  - How do we know when we are done with all operations on an object?

Deadlocks

- Possible for Ti to hold a lock Tj needs, and vice versa

	T1	T2	
	RA		
	WA		
		RB	
		WB	
T1 waits for T2 →	RB		
	WB		
		RA	← *T2 waits for T1*
		WA	

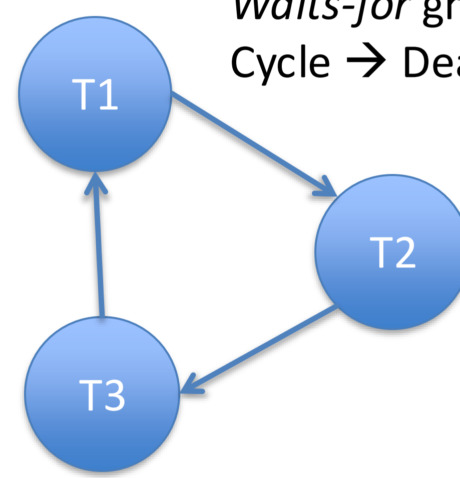
Waits-for graph
Cycle → Deadlock



Complex Deadlocks Are Possible

T1	T2	T3
RA		
WA		
		RC
	RB	
	WB	
		RA ← *T3 waits for T1*
		WA
T1 waits for T2 → RB		
WB		
	RC ← *T2 waits for T3*	
	WC	

Waits-for graph
Cycle → Deadlock



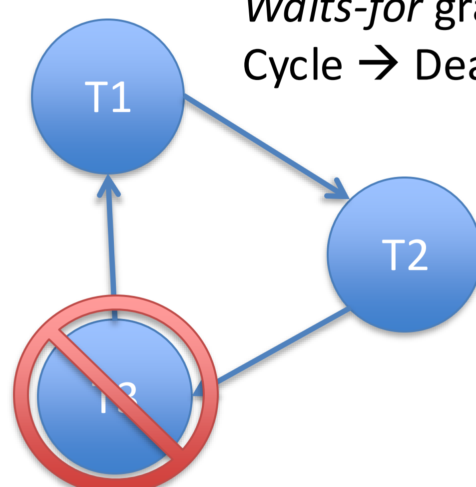
Resolving Deadlock

- Solution: abort one of the transactions
  - Recall: users can abort too

	T1	T2	
	RA		
	WA		
		RB	
		WB	
T1 waits for T2 →	RB		
	WB		
		RC	~~← *T2 waits for T3*~~
		WC	

Equivalent to T2 - T1

Waits-for graph
Cycle → Deadlock



Cascading Aborts

- Problem: if T1 aborts, and T2 read something T1 wrote, T2 also needs to abort

	T1	T2
	Xlock A	
	RA	
	WA	
	Xlock B	
	Rel A	
		Xlock A
		RA
		WA
If T1 aborts here →	RB	
T2 also needs to abort,	WB	
as it reads T1's write of A	Rel B	
		Xlock B
		RB
		WB
		Rel A,B

Can you think of a 2PL variant which neither requires deadlock detection nor has cascading aborts?

Strict Two-Phase Locking

- Can avoid cascading aborts by holding exclusive locks until end of transaction

- Ensures that transactions never read other transaction’s uncommitted data

Strict Two-Phase Locking Protocol

- Before every read, acquire a shared lock
- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock
- ~~Release locks only after last lock has been acquired, and ops on that object are finished~~
- Release *shared* locks only after last lock has been acquired, and ops on that object are finished
- Release *exclusive* locks only after the transaction commits
- Ensures cascadeless-ness

Problem: When is it OK to release?

- How does DBMS know a transaction no longer needs a lock?
- Difficult, since transactions can be issued interactively
- In practice, this means that all locks held until end of transaction
- This is called *rigorous two-phase locking*

Rigorous Two-Phase Locking Protocol

- Before every read, acquire a shared lock

- Before every write, acquire an exclusive lock (or "upgrade") a shared to an exclusive lock

- Release locks only after the transaction commits

- Ensures cascadeless-ness, and
- *Commit order = serialization order*

Next Lectures

- Optimistic concurrency control: Another protocol to achieve conflict serializability

- Nuances that arise with locking granularity