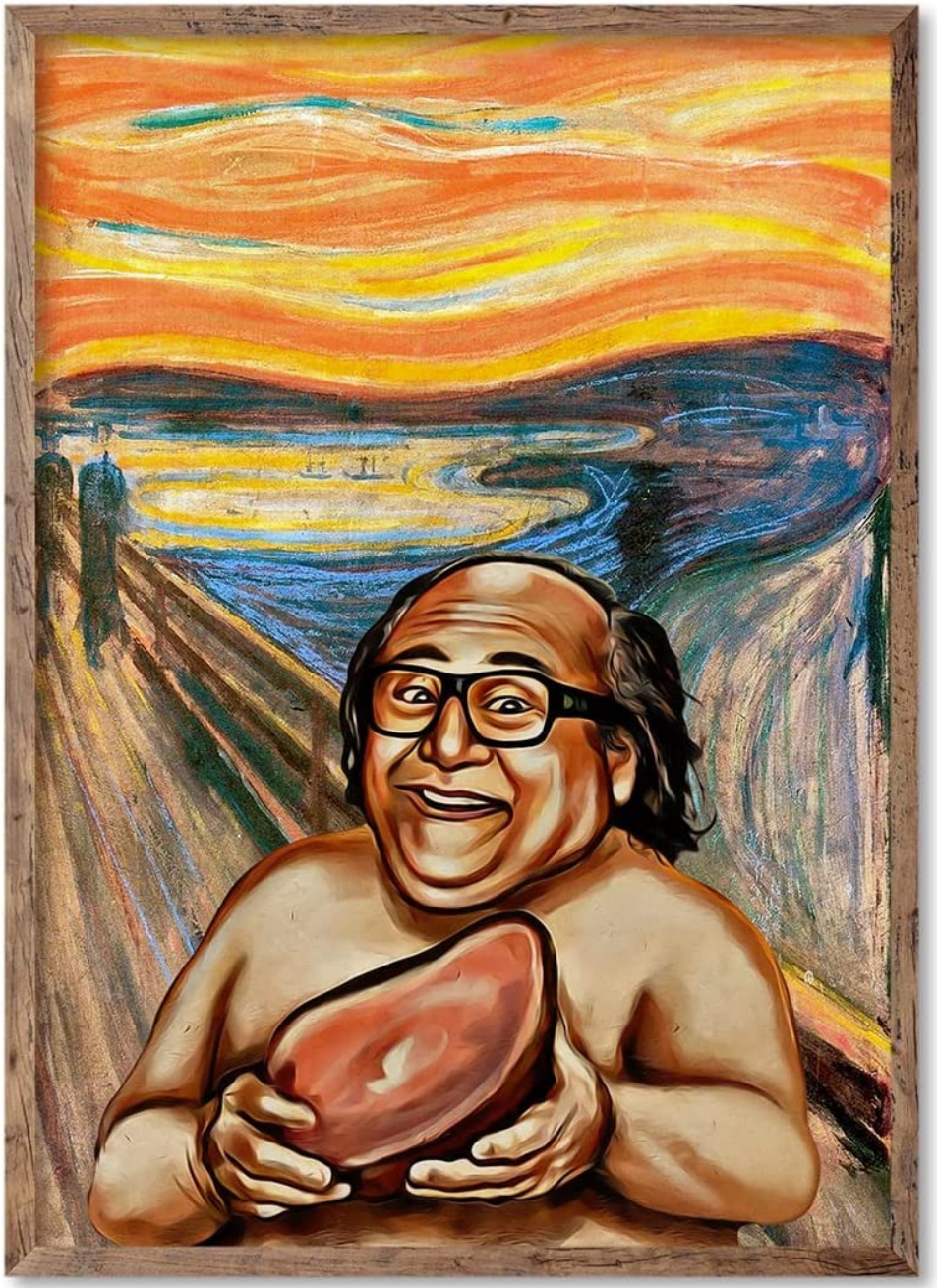


6.5830 Lecture 11 Two-phase Locking (recap) & Optimistic Concurrency Control

10/18/2022


JEOPARDY!
The Database Tour

Rules

- **Reverse Jeopardy: The answers are the answers**
- **Create groups of 3-5 people**
- **You have to answer as a group**
- **Correct answers give candy points.**
- **Groups willing to answer the question have to raise their hand**
- **Among all hand-raised groups the moderator randomly picks a group.**
- **At the discretion of the moderator the answer is deemed correct or wrong. If correct, the group can choose the next question. If wrong, another group can answer (but no candy)**
- **You have to pick questions from the top down**

Terminology	Optimistic Concurrency Control	2 Phase Locking	Other Concurrency Techniques
1 Candy	2 Candy	1 Candy	1 Candy
2 Candy	4 Candy	2 Candy	2 Candy
3 Candy	8 Candy	3 Candy	3 Candy
4 Candy	8 Candy	4 Candy	4 Candy
5 Candy	8 Candy	5 Candy	5 Candy

1 Candy

What is the connection between Atomicity, Isolation, and Durability with Concurrency Control, Logging?

Back

2 Candy

Today Sam and I almost dressed alike. Let's assume the process of dressing is done as part of a transaction, which ensures that we always dress differently.

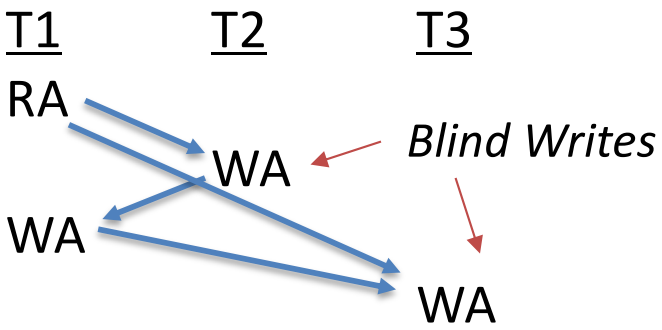
What ACID properties were violated?

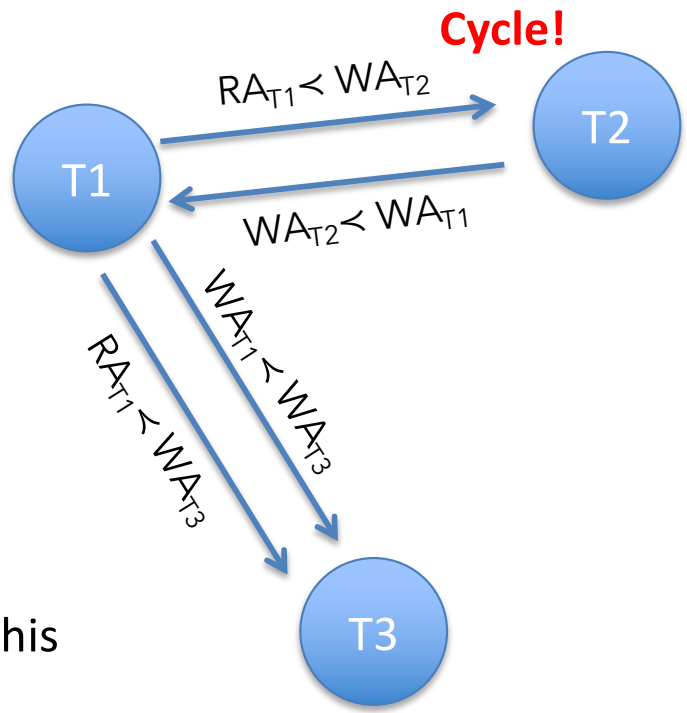
3 Candy

Name 2 types of serializability and give an example when one is valid under the definition but not the other?

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*

Back

4 Candy

If a system ensures Atomicity, Durability, and Isolation, is it by definition not also Consistent? Explain by means of an example why/why not?

Back

5 Candy

From the assigned reading from lecture 11:

WAL, LSN, Undo, Redo, physical, logical, physiological are all important terms for describing logging. What do they stand for?

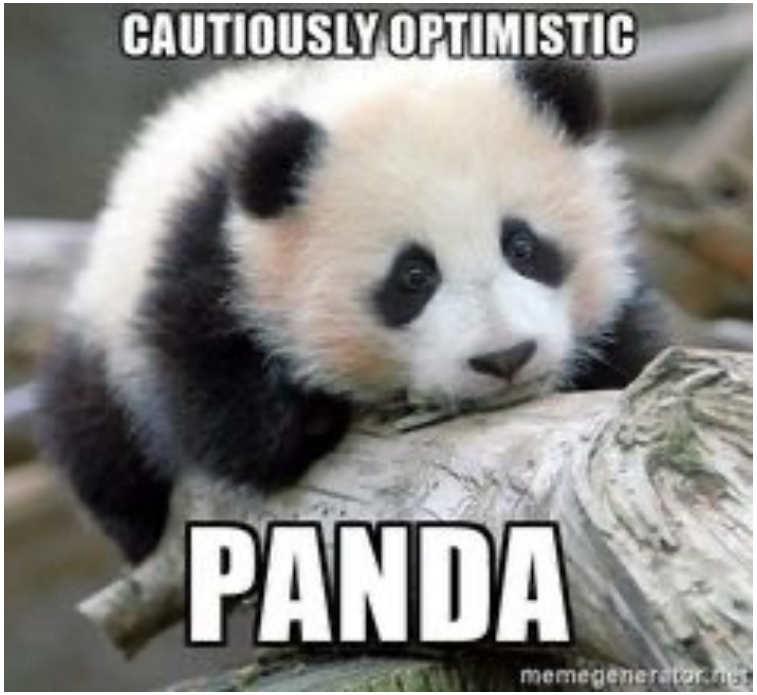
2 Candy

What is the core idea of optimistic concurrency control?

What are the pros and cons?

(You should know that based on the assigned reading :)

Optimistic Concurrency Control (OCC)



- Alternative to locking for isolation

- Approach:
  - Store writes in a per-transaction buffer
  - Track read and write sets
  - At commit, check if transaction conflicted with earlier (concurrent) transactions
  - Abort transactions that conflict
  - Install writes at end of transaction
- "Optimistic" in that it does not block, hopes to "get lucky" arrive in serial interleaving

Back

Tradeoff

- In OCC:
  - Never have to wait for locks
  - no deadlocks
- But...
  - Transactions that conflict often have to be restarted
  - **Transactions can "starve" -- e.g., be repeatedly restarted, never making progress**
- OCC will do better when the restart rate is low
  - (Less contention)
- Recent work on high performance transaction processing has focused on OCC because
  - OCC checks can be done between individual transactions
    - Unlike global shared lock table
  - Modern OCC systems obtain *insane* throughput (> 10M xactions / sec)

E.g., https://people.eecs.berkeley.edu/~wzheng/silo.pdf

4 Candy

What are the 3 phases of optimistic concurrency control?

OCC Implementation

- Divide transaction execution in 3 phases
  - **Read**: transaction executes on DB, stores local state
  - **Validate**: transaction checks if it can commit
  - **Write**: transaction writes state to DB

Read Phase

- Transactions execute, with updates affecting local copies of the data
- Build a list of data items that were read/written
  - *Read* and *write sets*
- Modify functions to read and write data from DB

OCC Write

```
twrite(object,value):
    if object not in write_set:  // never written, make copy
        m = read(object)
        copies[object] = m
        write_set = write_set U {object}
   write(copies[object], value)
```

By writing to local copies, we ensure dirty results aren't visible to other concurrent transactions

OCC Read

```
tread(object):
        read_set = read_set U {object};
        if object in write_set:
                return read(copies[object]);
        else:
                return read(object);
```

Allows us to read our own writes!

Back

Validation Phase

- How do we know whether a transaction can commit?
  - *Validation Rules*
  - Check concurrent transactions for conflicts
- How do we implement validation efficiently?
  - *Validation Algorithm*
- *But first…* How how do we order transactions?

8 Candy

Goal: assign transaction ids T1, … Tn, such that this is the serial equivalent orde.r

When do you assign the Transaction Identifier?

a) **At the beginning of the trx,**
b) **At the start of the validation phase**
c) **At the start of the write phase**
d) **At the end of the write phase**

Back

Transaction Identifier Assignment

- Goal: assign transaction ids T1, … Tn, such that this is the serial equivalent order
- When should we assign transaction identifiers?
- At start of read phase?
  - No! Would be “pessimistic” – don’t want to pre-assign the transaction order before transactions finish running
  - Long running transactions would have to commit before later short transactions
- Assign at end of read phase, just before validation starts

8 Candy

Under OCC there are 4 conditions. When Tj completes its read phase, require that for all Ti < Tj, one of the following conditions must be true for validation to succeed (Tj to commit) to ensure serializablity:

1) **Ti completes its write phase before Tj starts its read phase**
2) **W(Ti) does not intersect with R(Tj) or W(tj)** 
3) **W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.**
4) **W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.**
5) **W(Ti) does not intersect R(Tj) or W(Tj), and W(Tj) does not intersect R(Ti) [no conflicts]**

One of the conditions is not sufficient! Which one?

Validation Rules

XXX confusion about W and R as Read / write sets vs read / write phases

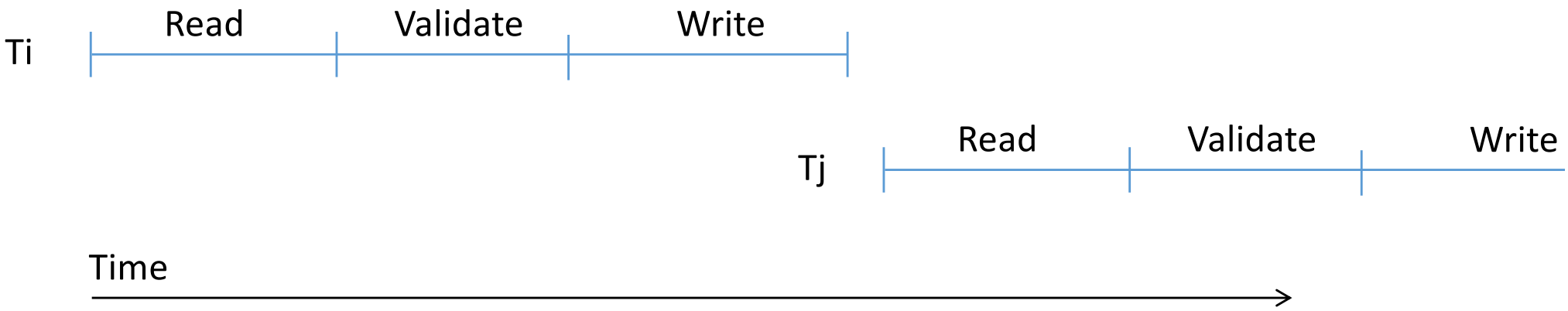
When Tj completes its read phase, require that for all Ti < Tj, one of the following conditions must be true for validation to succeed (Tj to commit):

1) Ti completes its write phase before Tj starts its read phase
2) W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.
3) W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.
4) W(Ti) does not intersect R(Tj) or W(Tj), and W(Tj) does not intersect R(Ti) [no conflicts]

These rules will ensure serializability, with Tj being ordered after Ti with respect to conflicts

Condition 1

Ti completes its write phase before Tj starts its read phase



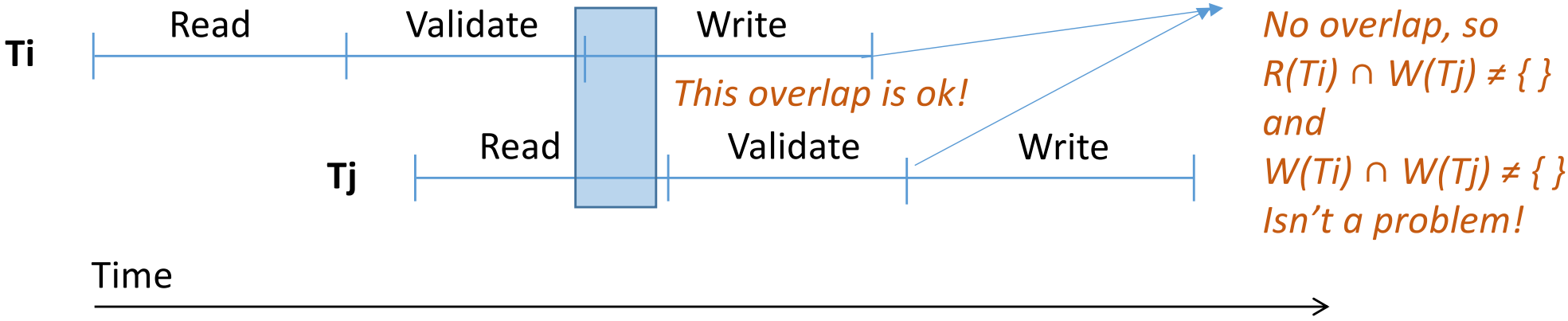
Don't overlap at all.

Condition 2

W(Ti) intersects W(Tj), i.e., Tj wrote something Ti wrote,
or
R(Ti) intersects W(Tj), i.e., Ti read something Tj wrote

W(Ti) does not intersect R(Tj), and Ti completes its write phase before Tj starts its write phase.

$W(Ti) \cap R(Tj) = \{\}$ $R(Ti) \cap W(Tj) \neq \{\}$ $W(Ti) \cap W(Tj) \neq \{\}$



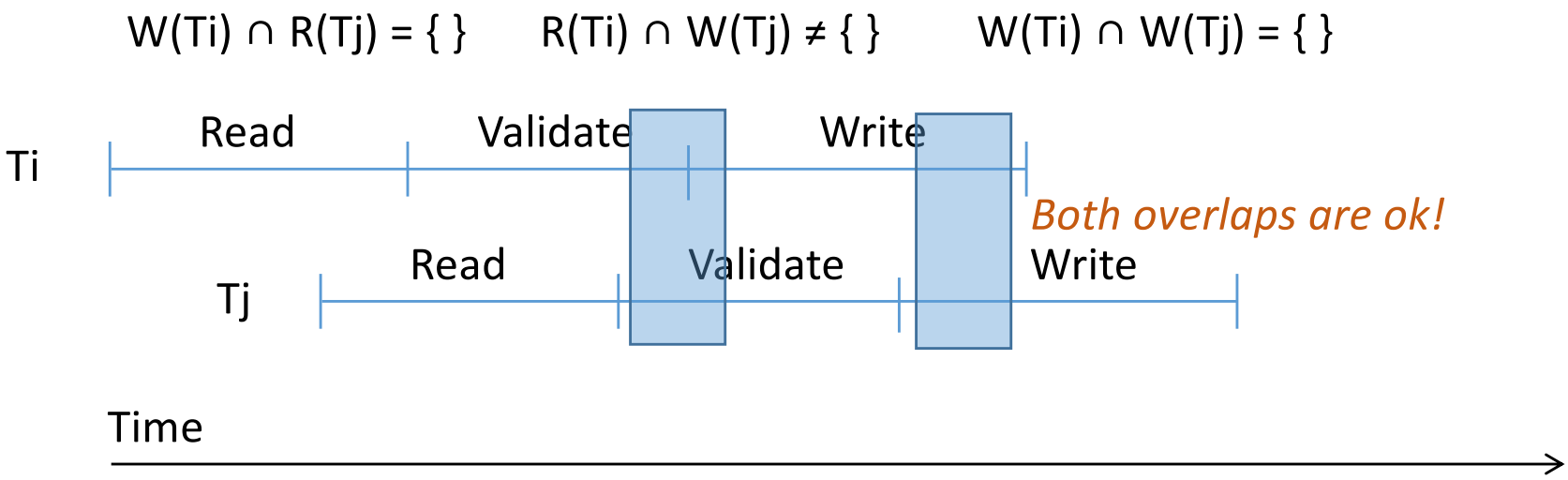
No overlap, so
$R(Ti) \cap W(Tj) \neq \{\}$
and
$W(Ti) \cap W(Tj) \neq \{\}$
Isn't a problem!

Tj doesn't read anything Ti wrote.
Anything Tj wrote that Ti also wrote will be installed afterwards.
Anything Ti read will not reflect Tjs writes

Condition 3

R(Ti) intersects W(Tj), i.e., Ti read something Tj wrote

W(Ti) does not intersect R(Tj) or W(Tj), and Ti completes its read phase before Tj completes its read phase.



Tj doesn't read or write anything Ti wrote (but Ti may read something Tj writes).

Ti definitely won't see any of Tj's writes, because it finishes reading before Tj starts validation, so Ti ordered before Tj.

Ti will always complete its read phase before Tj b/c xaction IDs assigned after read phase

Back

If no conditions apply, abort!

Restating previous rules, aborts required if:

1) $W(Ti) \cap R(Tj) \neq \{\}$, and Ti does not finish writing before Tj starts, Tj must abort, because Tj may have only seen some of what Ti wrote

or

2) $W(Ti) \cap (W(Tj) \cup R(Tj)) \neq \{\}$, and Tj overlaps with Ti validation or write phase, Tj must abort because it needs its writes to all appear after Ti's writes

Back

8 Candy

Validate Implementation

Validate Implementation

- Several different implementations designed to provide different levels of concurrency during writeback

- Transaction initialization:

```
tnc = 0; // current transaction id
void tbegin {
    read_set = new Set();
    write_set = new Set();
    start_tn = tnc; //the transaction that
                    //finished just before this
                    //one started
}
```

Serial Validation

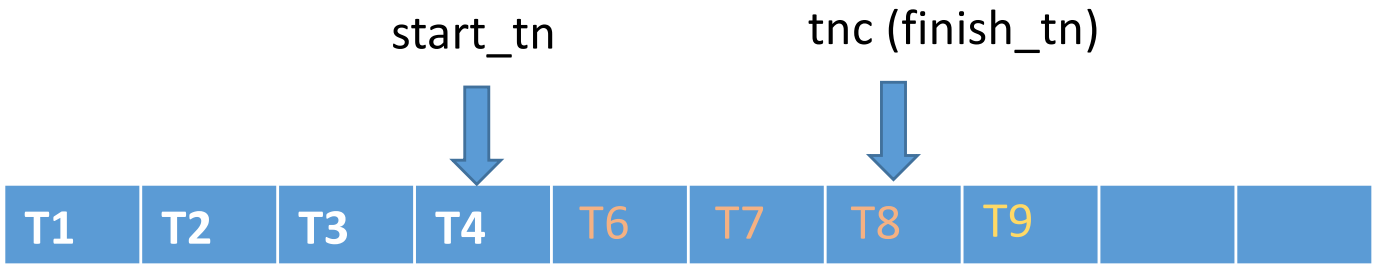
```
validateAndWrite(pastT[], start_tn, my_read_set, my_write_set)
{
    lock();
    int finish_tn = tnc;  //prior transaction
    bool valid = true;
    for(int t = start_tn + 1; t <= finish_tn; t++)
        if(pastT[t].write_set intersects with my_read_set)
            valid = false;
    if (valid) {
        write_phase();
        tnc = tnc+1;
        tn = tnc;
    }
    unlock();
}
```

1. W(Ti) ∩ R(Tj) ≠ { }, and Ti does not finish writing before Tj starts, Tj must abort
2. ~~W(Ti) ∩ (W(Tj) U R(Tj)) ≠ { }, and Tj overlaps with Ti validation or write phase, Tj must abort~~

2nd condition doesn't occur because if Ti completes its read phase before Tj, it will also complete its write phase before Tj.

Example

Last transaction that validated before this transaction started



Transactions that validated and wrote while this transaction was in read phase

Have to compare against T6, T7, and T8

Back

Which of the following transactions would serial validation allow to commit, assuming the transactions are concurrent and Ti completes its write phase before Tj starts its write phase ∀ i < j

Aborts required if:

1) W(Ti) ∩ R(Tj) ≠ { }, and Ti does not finish writing before Tj starts, Tj must abort, because Tj may have only some of what Ti wrote

or

2) W(Ti) ∩ W(Tj) ∩ R(Tj)) ≠ { } , and Tj overlaps with Ti validation or write phase, must abort because it needs writes to all appear after Ti's writes

A.

T1	T2
RA	RC
WA	WA
RB	
WB	

Blind write

B.

T1	T2	T3
RA	RB	RA
WA	WB	
		WC

C.

T1	T2
RA	RA
WA	
	WA

Back

1 Candy

Is the following schedule permitted by two-phase locking?

T1	T2	T3
READ A		
		READ A
	WRITE A	
		WRITE B
WRITE A		
	WRITE B	
COMMIT		
	COMMIT	
		COMMIT

Is this schedule view serializable? Is it conflict serializable?

Back

2 Candy

Which of the following schedules would rigorous 2PL permit?

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

T1	T2
RA	
WA	
	RC
	WC
RB	
WB	
COMMIT	
	RB
	WB
	COMMIT

T1	T2
RA	
WA	
	RB
	WB
	COMMIT
RB	
WB	
COMMIT	

T1 does not release lock on A until COMMIT (note under rigorous 2PL T2 would wait so this schedule would not happen)

3 Candy

Consider the following banking schema (a typical design)

account_nb	owner	status
account		
2	Karin	open
…	…	…

id	account_nb	description	amount
bank_trx			
3	2	Candy for Tim	$ -40
4	1	Ski pass	$ -50
…		…	…

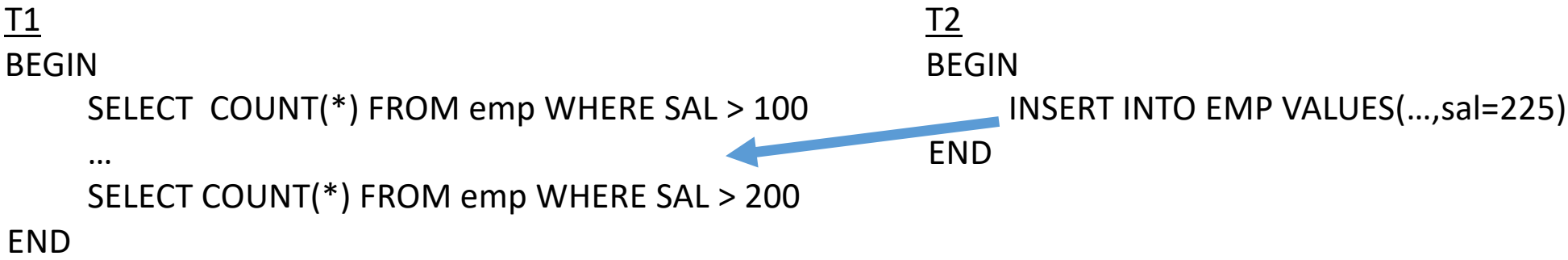
You have decided with your girl-/boyfriend to consolidate the accounts by closing ACCOUNT_NB=1 with the following trx:

```
acct_sum = SELECT SUM(amount) FROM bank_trx WHERE account_nb = 1
INSERT (100, 2, "Transfer from Acct 1", acct_sum) INTO bank_trx
INSERT (101, 1, "Transfer to Acct 2", -acct_sum) INTO bank_trx
UPDATE account SET status=closed WHERE account_nb = 1
```

Assuming transfers always check the account status, why is 2-Phase Locking from class not sufficient to ensure that the balance of account 1 is 0 after closing it? How would you prevent the problem?

Final Wrinkle: Phantoms

- T1 scans a range; T2 later inserts into that range
- If T1 scans the range again, it will see a new value

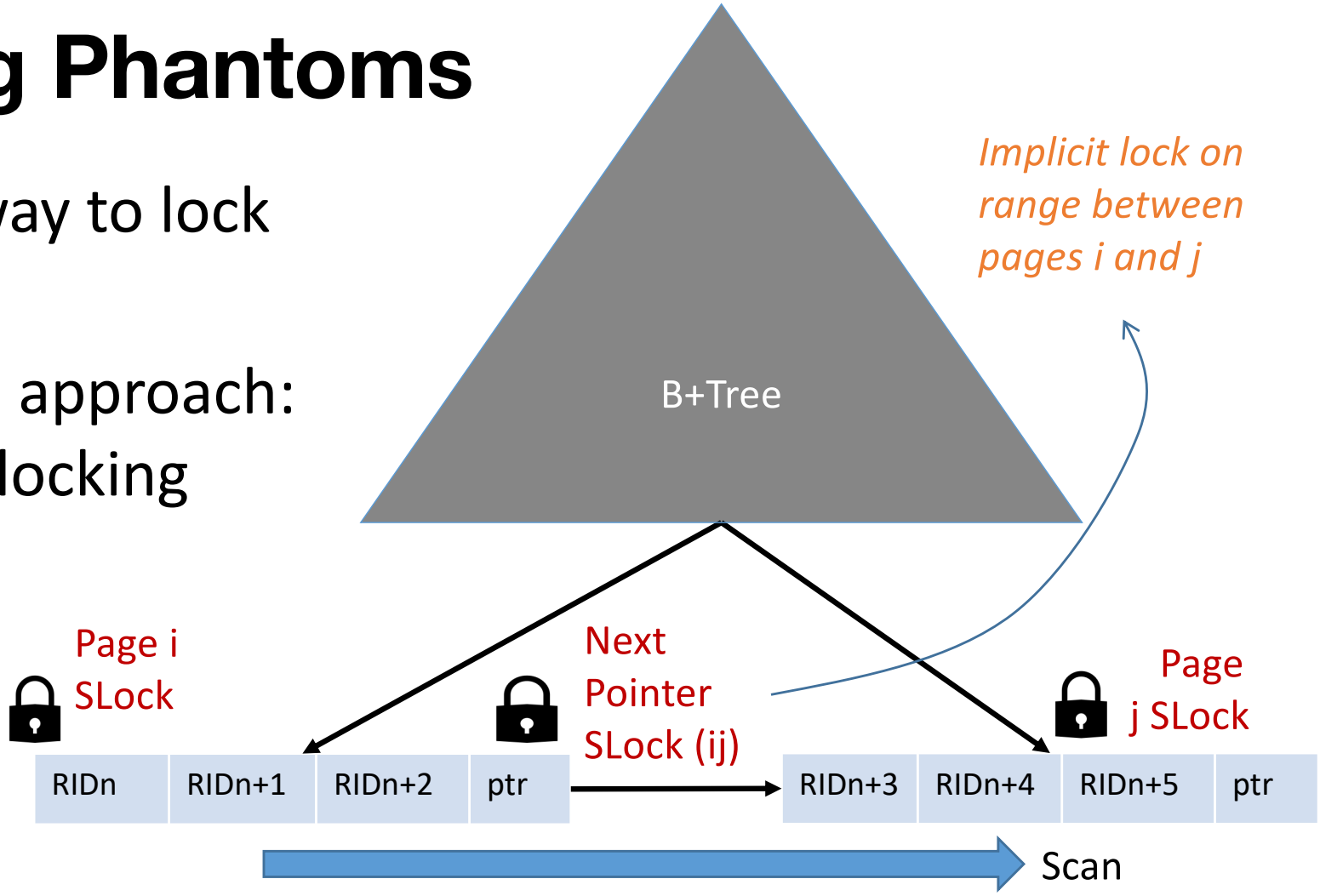


If we are just locking, e.g., records, this insertion would be allowed in all 2PL algos we have studied, but is not serializable (since this couldn't happen in a serial execution).

Back

Solving Phantoms

- Need a way to lock ranges
- Common approach: next key locking



On insert(val), Xlock ij next pointer if val> max(page i) and < min(page(j))

Only works for ranges with indexes
For unindexed tables, must read the whole table, so just use a table lock
More details next lecture!

4 Candy

How would you implement 2PL and prevent Phantoms in GoDB without a Btree?

Back

Implementing 2PL

- **GoDB**: Lock Table
  - Buffer pool maintains a table of locks per page
  - Transactions acquire locks on reads/writes of pages
  - Release locks at commit

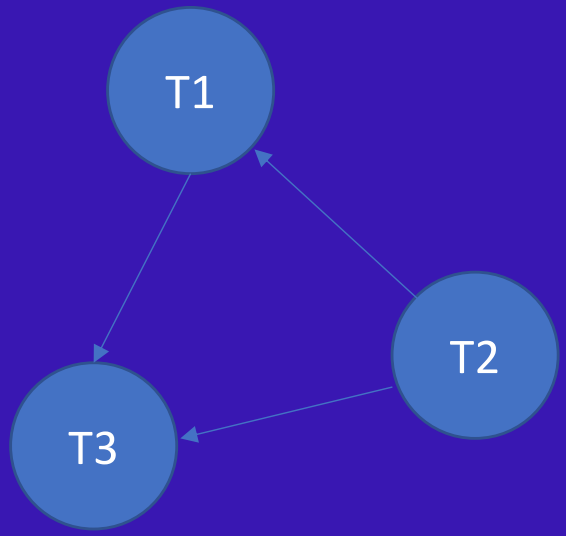
- Access to lock table will need to be synchronized

Back

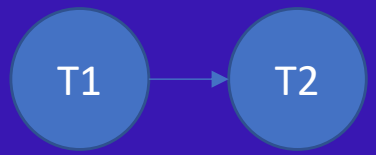
5 Candy

What is the conflict graph for this schedule, and is it serializable?

T1	T2	T3
READ A		
	READ A	
	WRITE B	
		READ B
WRITE A		
READ B		
		WRITE B



For an operation pair that conflicts in T1 and T2, the operation happens first in T1



1 Candy

What if Serializability isn't needed?

What If Serializability Isn't Needed?

- E.g., application only needs to read committed data
- Databases provide different isolation levels
  - READ UNCOMMITTED
    - Ok to read other transaction's dirty data
  - READ COMMITTED
    - Only read committed values
  - REPEATABLE READS
    - If R1 read A=x, R2 will read A=x ∀ A
- Many database systems default to READ COMMITTED

READ UNCOMMITTED w/ Locking

- If OK reading uncommitted data, no need to check if records that are read are locked
- However, to prevent other transactions from seeing dirty data, need to hold write locks for the duration of the transaction
- May be OK if, e.g., just reporting some statistic, like number of users or views

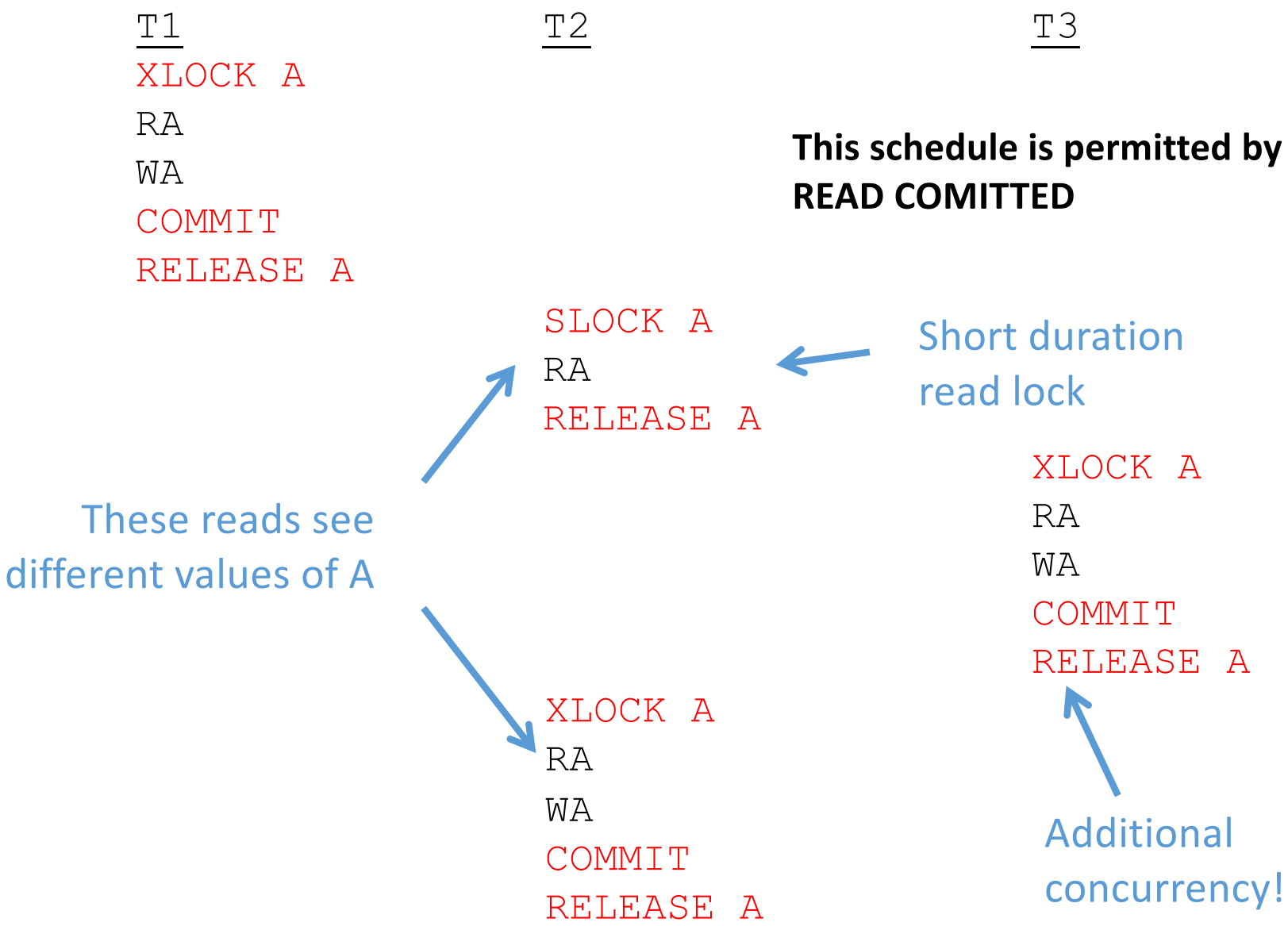
READ COMMITTED w/ Locking

- To ensure that a transaction only reads committed values, need to acquire locks before reading
  - If all other transactions hold write locks (as in READ UNCOMMITTED), it will never read a dirty value
- Since we doesn't care about always reading the same value, OK to release locks after a value is read
- As in READ UNCOMMITTED, write locks still need to be held for the duration of the transaction

Is this schedule permitted under (a) read uncommitted, (b) read committed, (c) repeatable read

T1	T2	T3
RA		
WA		
COMMIT		
	RA	
		RA
		WA
		COMMIT
	RA	
	WA	
	COMMIT	

READ COMMITTED Example



Back

REPEATABLE READ w/ Locking

- If we want to always read the same value, need to hold read locks for transaction duration
- So how is this different from SERIALIZABLE?
- SERIALIZABLE also needs to prevent phantoms

2 Candy

Multi-version concurrency control (MVCC) / Snapshot Isolation (SI)

REPEATABLE READ vs SERIALIZABLE

- Some systems, e.g., Postgres implement REPEATABLE READ through a different mechanism based on database snapshots taken at the start of transaction
  - Called “multiversion concurrency control” – yet another way of achieving isolation!

- This has other problems besides phantoms – so called “read skew anomalies”
  - See: https://www.cockroachlabs.com/blog/what-write-skew-looks-like/

Snapshot Isolation

- When a TA starts it receives a timestamp, T.
- All reads are carried out *as of* the DB version of T.
  - Need to keep historic versions of all objects!!!
- All writes are carried out in a separate buffer.
  - Writes only become visible after a commit.
- When TA commits, DBMS checks for conflicts
  - Abort TA1 with timestamp T1 if exists TA2 such that
    - TA2 committed after T1 and before TA1
    - TA1 and TA2 updated the same object

UNDER SI
A) Does T2 abort or commit?
B) Does T3 abort or commit?

T1	T2	T3
		R(Y)
W(Y) Commit		
	Start R(X) R(Y)	
		W(X) W(Z) Commit/Abort
	R(Z) R(Y) W(X) Abort?/Commit?	

Back

Is the following schedule valid under SI

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
		R(Y)->v0
W(Y := v1) Commit		
	Start R(X) → v0 R(Y)→ v1	
		W(X:=v2) W(Z:=v1) Commit
	R(Z) → v0 R(Y) → v1 W(X:=v3) Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

Back

3 Candy

For what type of workloads should you use 2PL for what type of workloads SI?

4 Candy

Create a real-world-inspired example, (e.g., from banking, hotel booking, …) for which two transactions executed with Snapshot Isolation guarantees would violate serializibility.

Back

- Observation: **Snapshot isolation does not prevent Write-Read conflicts, since it doesn't check whether it read something another transaction wrote**
- **This leads to so-called write-skew, i.e.:**

T1	T2
RX	
	RY
WY	
	WX

- **Neither transaction saw the other's write; this would not be permitted under serializability**

Real world example:

- **Employee X and employee Y are signing up for shifts**
- **They hate each other and don't want to work together**
- **EY checks that EX isn't working (RX)**
- **EX checks that EY isn't working (RY)**
- **Both update their schedule to work on the same day**

Back

5 Candy

Does SI isolation have the phantom problem?

Back

Under 2-Phase Locking, the following locks got requested from the Lock Manager. Is there a deadlock? Can you think of a single change to create/resolve the deadlock?

T1	T2	T3
RA	RB	RC
WB	RC	RA

time	Type	TID
t1	Req. ReadLock A	T1
t2	Req. ReadLock B	T2
t3	Req. ReadLock C	T3
t4	Req. ReadLock A	T3
t5	Req. WriteLock B	T1
t6	Req. ReadLock C	T2