

# TRANSACTION MANAGEMENT II

---

*CS 564- Fall 2018*

---

*ACKs: Jeff Naughton, Jignesh Patel, AnHai Doan*

---

# WHAT IS THIS LECTURE ABOUT?

---

- Transaction (TXN) management
- **ACID** properties
  - atomicity
  - consistency
  - isolation
  - durability
- Logging
- Scheduling & locking

---

# ACID PROPERTIES: RECAP

---

**Atomicity:** all actions in the TXN happen, or none happen

**Consistency:** a database in a consistent state will remain in a consistent state after the TXN

**Isolation:** the execution of one TXN is isolated from other (possibly interleaved) TXNs

**Durability:** once a TXN **commits**, its effects must persist

---

# CONCURRENCY

---

# CONCURRENCY

---

- The DBMS runs multiple TXNs concurrently
- To achieve better performance, **interleaving** the operations of the TXNs is critical
  - possibly slow TXNs
  - CPU/IO overlap
- But interleaving can lead to problems!



Remember: we must guarantee **isolation & consistency!**

# EXAMPLE

**T1:** *transfer \$100 from A to B*

```
BEGIN TRANSACTION ;  
  UPDATE account  
    SET balance = balance - 100  
    WHERE account_name = A;  
  UPDATE account  
    SET balance = balance + 100  
    WHERE account_name = B;  
COMMIT ;
```

**T2:** *add 10% interest to both accounts*

```
BEGIN TRANSACTION ;  
  UPDATE account  
    SET balance = balance * 1.1  
COMMIT ;
```

Let's see how the DBMS can schedule the 2 transactions

# EXAMPLE

First run T1, then run T2

<i>T1</i>	<i>T2</i>
$A \leftarrow A - 100$	
$B \leftarrow B + 100$	
	$A \leftarrow A * 1.1$
	$B \leftarrow B * 1.1$

time

Beginning

- $A = 200, B = 100$

End

- $A = 110, B = 220$

This is called a **serial** schedule

# EXAMPLE

First run T2, then run T1

<i>T1</i>	<i>T2</i>
	$A \leftarrow A * 1.1$
	$B \leftarrow B * 1.1$
$A \leftarrow A - 100$	
$B \leftarrow B + 100$	

time

Beginning

- $A = 200, B = 100$

End

- $A = 120, B = 210$

This is also a serial schedule



# EXAMPLE

Interleaving the operations of T1 and T2

<i>T1</i>	<i>T2</i>
	$A \leftarrow A * 1.1$
$A \leftarrow A - 100$	
	$B \leftarrow B * 1.1$
$B \leftarrow B + 100$	

time

Beginning

- $A = 200, B = 100$

End

- $A = 120, B = 210$

Same result as if we run serially T2 and then T1! This is called a **serializable** schedule

# EXAMPLE

Different interleaving of the operations of T1 and T2

<i>T1</i>	<i>T2</i>
	$A \leftarrow A * 1.1$
$A \leftarrow A - 100$	
$B \leftarrow B + 100$	
	$B \leftarrow B * 1.1$

time

Beginning

- $A = 200, B = 100$

End

- $A = 120, B = 220$

Different result from both serial schedules!  
This is called a **not serializable** schedule

# SCHEDULES: DEFINITIONS

**Schedule**: an interleaving of actions from a set of TXNs, where the actions of any TXN are in the original order

**Serial schedule**: a schedule where there is no interleaving of actions from different TXNs

**Equivalent schedules**: two schedules are equivalent if *for every* database state, they will have the same effect

**Serializable schedule**: a schedule that is equivalent to *some* serial schedule

Note: we assume that all TXNs commit in the schedules!

# THE DBMS'S VIEW OF THE SCHEDULE

<i>T1</i>	<i>T2</i>
	$A \leftarrow A * 1.1$
$A \leftarrow A - 100$	
$B \leftarrow B + 100$	
	$B \leftarrow B * 1.1$

time

Each action is a read (**R**)  
followed by a write (**W**)

<i>T1</i>	<i>T2</i>
	R(A)
	W(A)
R(A)	
W(A)	
R(B)	
W(B)	
	R(B)
	W(B)

# CONFLICTS IN SCHEDULES

Two actions **conflict** if they are part of different TXNs, involve the same variable, and at least one of them is a write

- Write-Read conflict
- Read-Write conflict
- Write-Write conflict

A conflict does not always lead to a problem when interleaving!

# CONFLICTS VS ANOMALIES

**Conflicts** help us characterize different schedules

- present in both “good” and “bad” schedules

**Anomalies** are instances where isolation and/or consistency is broken because of a “bad” schedule

- we often characterize different anomaly types by what types of conflicts predicated them

# DIRTY READ

<i>T1</i>	<i>T2</i>
	W(A)
R(B)	
R(A)	
Commit	
	W(C)

time ↓

A **dirty read** occurs when a TXN reads data that was modified by a not yet committed TXN

- in the example, T1 reads A, which was previously modified by T2
- occurs because of a W-R conflict!



If T2 aborts, this will lead to inconsistency!

# UNREPEATABLE READ

<i>T1</i>	<i>T2</i>
	R(A)
W(A)	
R(B)	
Commit	
	R(A)

time

An unrepeatable read occurs when a TXN reads data twice, but in between the data was modified by another TXN

- in the example, T2 reads A, T1 then modifies T1, and T2 reads again
- occurs because of a R-W conflict!



# OVERWRITING UNCOMMITTED DATA

<i>T1</i>	<i>T2</i>
	W(A)
W(A)	
W(B)	
Commit	
	W(B)

time

This occurs when a TXN overwrites the data of an uncommitted TXN

- in the example, the last version of A and B would not be consistent with any serial schedule
- occurs because of a W-W conflict!

---

# CONFLICT SERIALIZABILITY

---

# CONFLICT SERIALIZABILITY


- Two schedules are **conflict equivalent** if:
  - they involve *the same actions of the same TXNs*
  - every *pair of conflicting actions* of two TXNs are *ordered in the same way*
- A schedule is **conflict serializable** if it is *conflict equivalent* to *some* serial schedule
- This provides us with a way to distinguish “good” from “bad” schedules

**Conflict serializable  $\Rightarrow$  serializable**

So if we have conflict serializable, we have consistency & isolation


# EXAMPLE

<i>T1</i>	<i>T2</i>
	R(A)
	W(A)
	R(B)
	W(B)
R(A)	
W(A)	
R(B)	
W(B)	



- In both, W(A) in T2 comes before R(A) in T1
- The same happens with all other pairs of conflicting actions
- Since the left schedule is serial, the right schedule is **conflict serializable**!

<i>T1</i>	<i>T2</i>
	R(A)
	W(A)
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	



# EXAMPLE

<i>T1</i>	<i>T2</i>
	R(A)
	W(A)
	R(B)
	W(B)
R(A)	
W(A)	
R(B)	
W(B)	

- The order has changed now!
- The two schedules are not conflict equivalent
- We still need to check all other serial schedules!

<i>T1</i>	<i>T2</i>
	R(A)
R(A)	
W(A)	
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

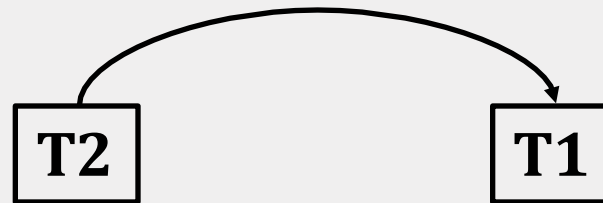
# THE CONFLICT GRAPH

---

- The conflict graph looks at conflicts at the transaction level
- the nodes are TXNs
- there is an edge from  $T_i$  to  $T_j$  *if any actions in  $T_i$  precede and conflict with any actions in  $T_j$*

# THE CONFLICT GRAPH

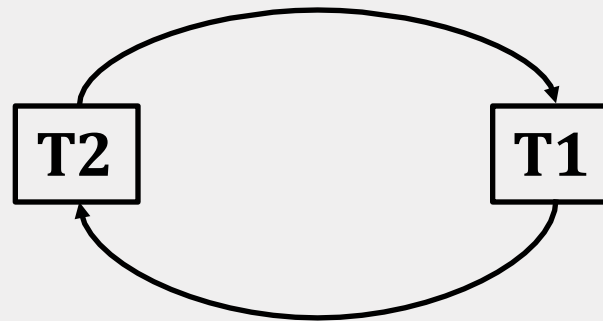
<i>T1</i>	<i>T2</i>
	R(A)
	W(A)
R(A)	
W(A)	
	R(B)
	W(B)
R(B)	
W(B)	



- Since W(A) in T2 is before R(A) in T1, we add an edge from T2 to T1
- There is no edge from T1 to T2 in this case!

# THE CONFLICT GRAPH

<i>T1</i>	<i>T2</i>
	R(A)
R(A)	
W(A)	
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	



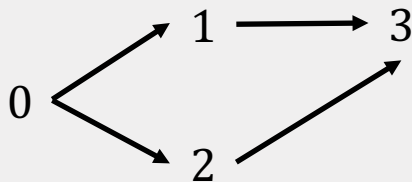
- Since R(A) in T1 is before W(A) in T2, we add an edge from T1 to T2
- Since W(B) in T2 is before R(B) in T1, we also add an edge from T2 to T1



# THE CONFLICT GRAPH: THEOREM

**Theorem:** a schedule is conflict serializable if and only if its conflict graph is **acyclic** (i.e. it has no directed cycles)

- A **topological ordering** of a directed graph is a linear ordering of its vertices that respects all the directed edges
- A **directed acyclic graph** (DAG) always has one or more topological orderings
  - if there are cycles, there exists no such ordering!

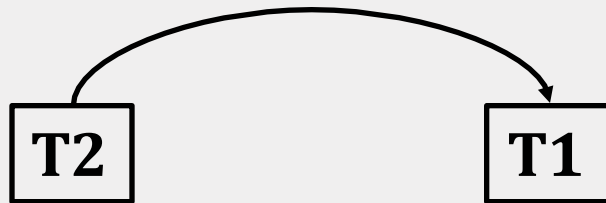


There are 2 possible topological orderings:

- 0, 2, 1, 3
- 0, 1, 2, 3

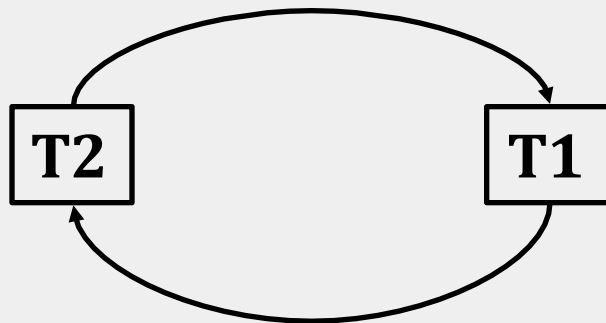
# THE CONFLICT GRAPH

- In the conflict graph, a topological ordering of the nodes corresponds to a **serial ordering** of TXNs (serial schedule)
- Thus an **acyclic** conflict graph  $\rightarrow$  conflict serializable!



**top ordering:** T2, T1

this is conflict equivalent to a serial schedule with first T2, then T1



there is a cycle, so no topological ordering  
not conflict serializable!

---

# LOCKING

---

# LOCKING

- Locking is a technique for **concurrency control**
- Lock information maintained by a *lock manager*:
  - stores (TID, RID, Mode) triples
  - mode is either Shared (S) or Exclusive (X)

	--	S	X
--	✓	✓	✓
S	✓	✓	
X	✓		

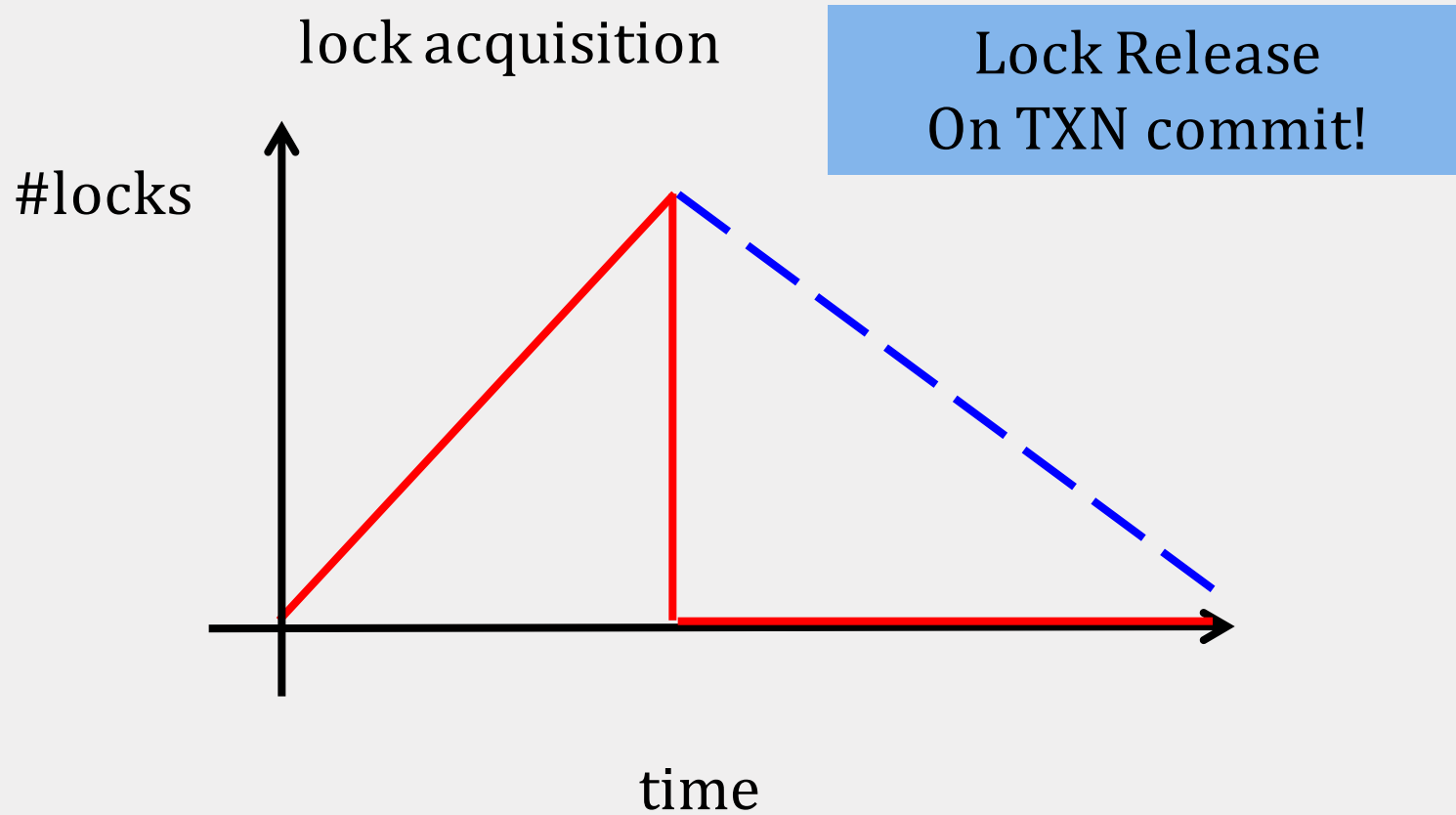
- If a transaction cannot get a lock, it has to wait in a queue

# STRICT 2 PHASE LOCKING

- Each transaction must obtain a **S** lock on object before reading, and an **X** lock on object before writing
- If a transaction holds an **X** lock on an object, no other transaction can get a lock (S or X) on that object
- All locks held by a transaction are released only when the transaction completes

Strict 2PL guarantees **conflict serializability!**

# STRICT 2PL: FIGURE



# DEADLOCKS

---

- If a schedule follows strict 2PL and locking, it is conflict serializable
  - and thus serializable
  - and thus maintains isolation & consistency!
- Not all serializable schedules are allowed by strict 2PL
- But running a strict 2PL protocol has some issues!

# STRICT 2PL

---

- If a schedule follows strict 2PL and locking, it is conflict serializable
  - and thus serializable
  - and thus maintains isolation & consistency!
- Not all serializable schedules are allowed by strict 2PL
- But running a strict 2PL protocol has some issues!



# DEADLOCKS

<i>T1</i>	<i>T2</i>
R(B)	
W(B)	
	R(A)
	W(A)
R(A)	
	R(B)

T1 gets an X-lock on B

T2 gets an X-lock on A

T1 wants to read A, but has to wait...

T2 wants to read B, but also has to wait...

We now have a **deadlock!**

# DEADLOCKS

---

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

---

# PERFORMANCE OF LOCKING

---

- Locks have a performance penalty:
  - **blocked** actions
  - **aborted** transactions
- Because of blocking, we can not increase forever the throughput of transactions
- At the point where the throughput cannot increase, we say that the system **thrashes**

---

# TRANSACTIONS IN SQL


---

# TRANSACTIONS IN SQL

---

- What object should we lock?


```
SELECT COUNT(*)  
FROM   Employee  
WHERE  age = 20 ;
```

- We can apply locking at different **granularities**:
  - lock the whole table Employee
  - lock only the rows with age = 20 

# TRANSACTIONS IN SQL

---

Transaction characteristics:

- Access mode: READ ONLY, READ WRITE
- Isolation level
  - **Serializable**: default (Strict 2PL) 
  - **Repeatable reads**: (R/W locks, but phantom can occur)
    - Read only committed records
    - Between two reads by the same transaction, no updates by another transaction
  - **Read committed** (W locks longterm, R locks shortterm)
    - Read only committed records
  - **Read uncommitted** (only reads, no locks)