Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo

Lecture#21: Concurrency Control – Part 1

(R&G ch. 17)

Last Class

- Introduction to Transactions
- ACID
- Concurrency Control
- Crash Recovery

Last Class

- For **Isolation** property, serial execution of transactions is safe but slow
  - We want to find schedules equivalent to serial execution but allow interleaving.
- The way the DBMS does this is with its ***concurrency control*** protocol.

Today's Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities

Formal Properties of Schedules

- **Serializable Schedule:** A schedule that is *equivalent* to some serial execution of the transactions.
- Note: If each transaction preserves consistency, every serializable schedule preserves consistency.

Example

T1

```
BEGIN
A=A+100
B=B–100
COMMIT
```

T2

```
BEGIN
A=A*1.06
B=B*1.06
COMMIT
```

- Consider two txns:
  - T1 transfers $100 from B's account to A's
  - T2 credits both accounts with 6% interest.
- Assume at first A and B each have $1000.

Interleaving Example (Good)

TIME

Schedule

T1	T2
BEGIN	
A=A+100	
	BEGIN
	A=A*1.06
B=B–100	
COMMIT	
	B=B*1.06
	COMMIT

A=1166, B=954

≡

Schedule

T1	T2
BEGIN	
A=A+100	
B=B–100	
COMMIT	
	BEGIN
	A=A*1.06
	B=B*1.06
	COMMIT

A=1166, B=954

Interleaving Example (Bad)

TIME

Schedule

T1	T2
BEGIN	
A=A+100	
	BEGIN
	A=A*1.06
	B=B*1.06
	COMMIT
B=B–100	
COMMIT	

A=1166, B=960

$\neq$

A=1166, B=954

or

A=1160, B=960

The bank lost $6!

Formal Properties of Schedules

- There are different levels of serializability:
  - **Conflict Serializability**
  - **View Serializability**

All DBMSs support this.

This is harder but allows for more concurrency.

Nobody does this.

Conflicting Operations

- We need a formal notion of equivalence that can be implemented efficiently…
  - Base it on the notion of “conflicting” operations

- Definition: Two operations conflict if:
  - They are by different transactions,
  - They are on the same object and at least one of them is a write.

Conflict Serializable Schedules

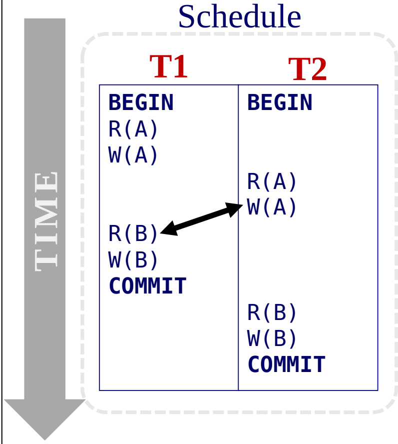
- Two schedules are ***conflict equivalent*** iff:
  - They involve the same actions of the same transactions, and
  - Every pair of conflicting actions is ordered the same way.
- Schedule S is ***conflict serializable*** if:
  - S is conflict equivalent to some serial schedule.

Conflict Serializability Intuition

- Schedule S is ***conflict serializable*** if:
  - You are able to transform S into a serial schedule by swapping consecutive non-conflicting operations of different transactions.

Conflict Serializability Intuition

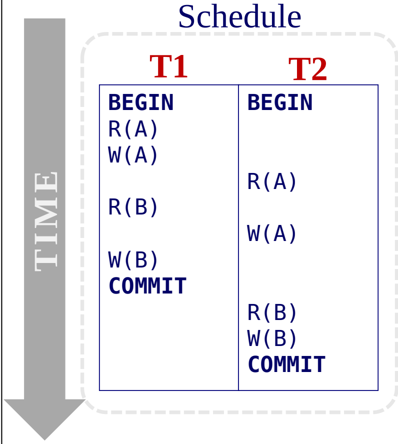
Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

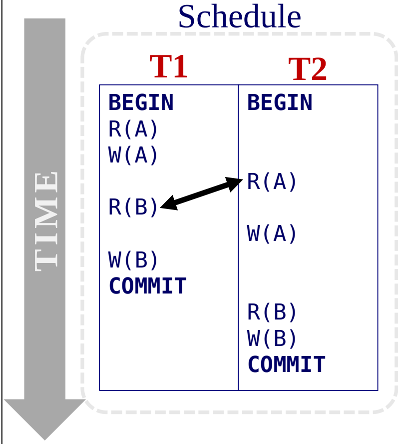
Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

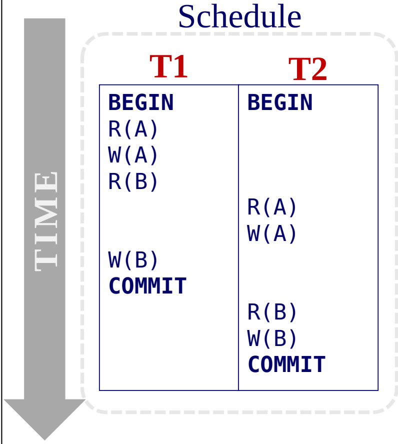
Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

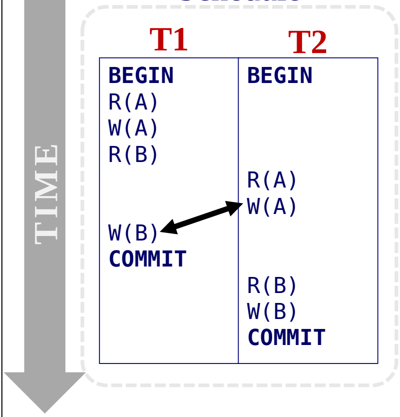
Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

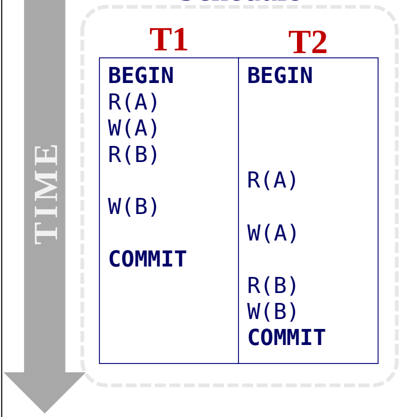
Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

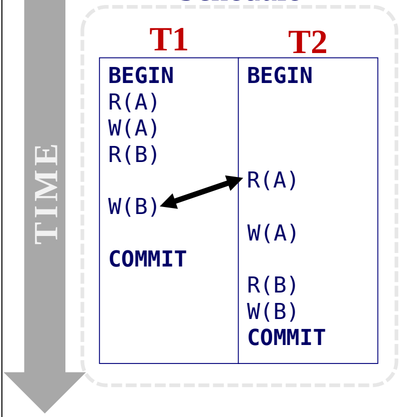
Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

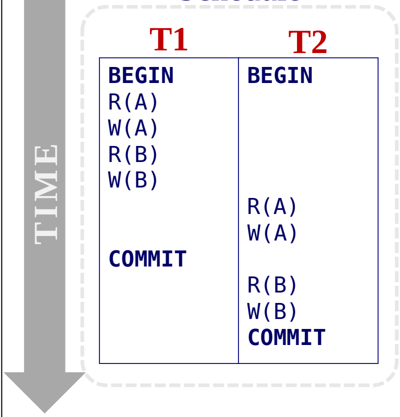
Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition

TIME

Schedule

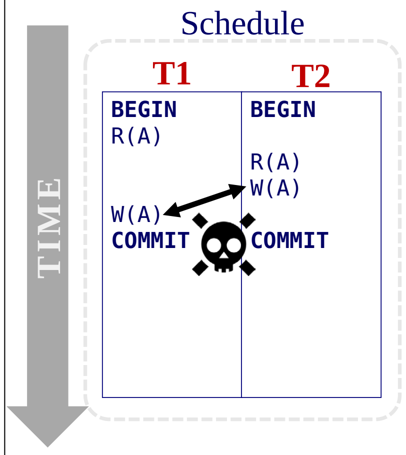
T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

≡

Serial Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	**BEGIN**
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

Conflict Serializability Intuition



Schedule

T1	T2
BEGIN	**BEGIN**
R(A)	
	R(A)
	W(A)
W(A)	
COMMIT	**COMMIT**

≠

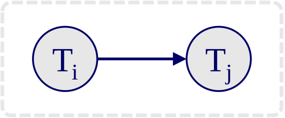
Serial Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
COMMIT	
	BEGIN
	R(A)
	W(A)
	COMMIT

Serializability

- **Q:** Are there any faster algorithms to figure this out other than transposing operations?

Dependency Graphs

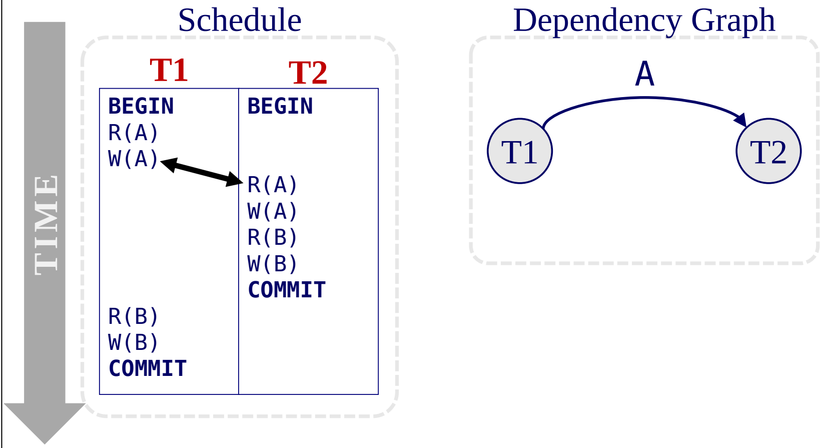


- One node per txn.
- Edge from T_i to T_j if:
  - An operation O_i of T_i conflicts with an operation O_j of T_j and
  - O_i appears earlier in the schedule than O_j.
- Also known as a “precedence graph”

Dependency Graphs

- **Theorem:** A schedule is *conflict serializable* if and only if its dependency graph is acyclic.

Example #1

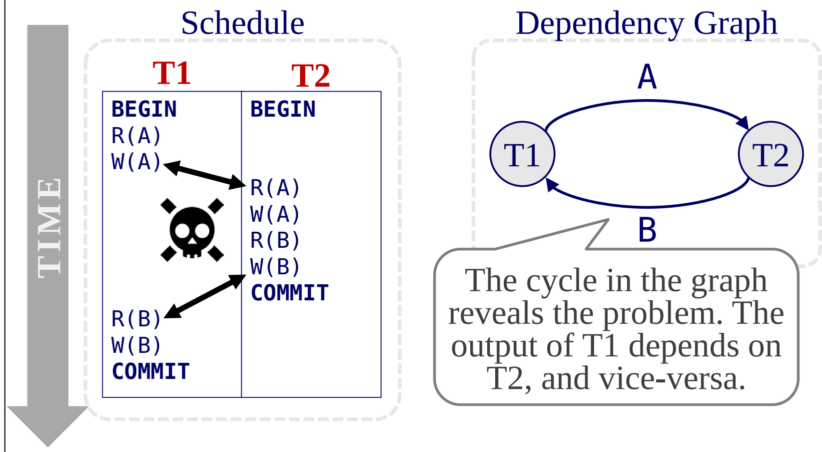


Schedule

T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph

Example #1



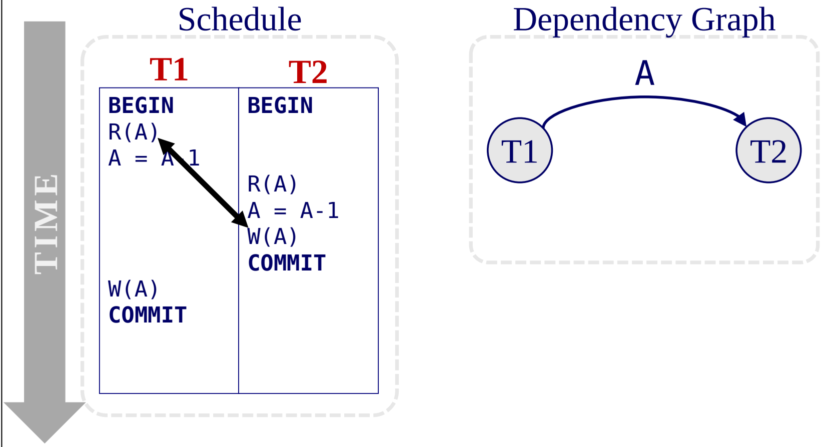
Schedule

T1	T2
BEGIN	**BEGIN**
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT
R(B)	
W(B)	
COMMIT	

Dependency Graph

The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Example #2 – Lost Update



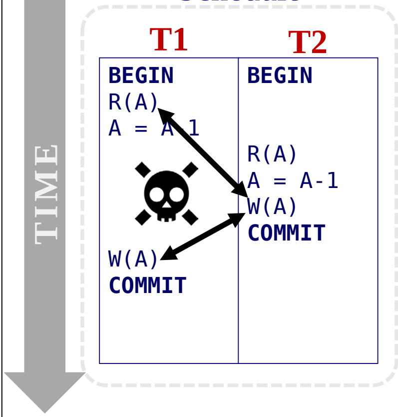
Schedule

T1	T2
BEGIN	**BEGIN**
R(A)	
A = A-1	
	R(A)
	A = A-1
	W(A)
	COMMIT
W(A)	
COMMIT	

Dependency Graph

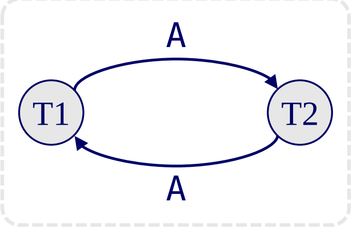
Example #2 – Lost Update

Schedule



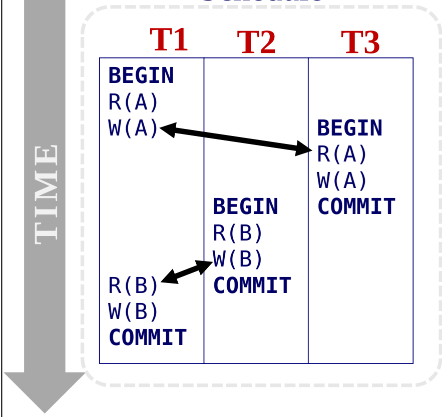
T1	T2
BEGIN	**BEGIN**
R(A)	
A = A-1	
	R(A)
	A = A-1
	W(A)
	COMMIT
W(A)	
COMMIT	

Dependency Graph



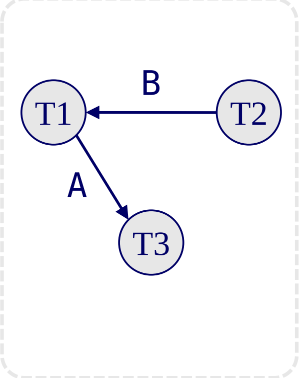
Example #3 – Threesome

Schedule



T1	T2	T3
BEGIN		
R(A)		
W(A)		**BEGIN**
		R(A)
		W(A)
	BEGIN	**COMMIT**
	R(B)	
	W(B)	
R(B)	**COMMIT**	
W(B)		
COMMIT		

Dependency Graph

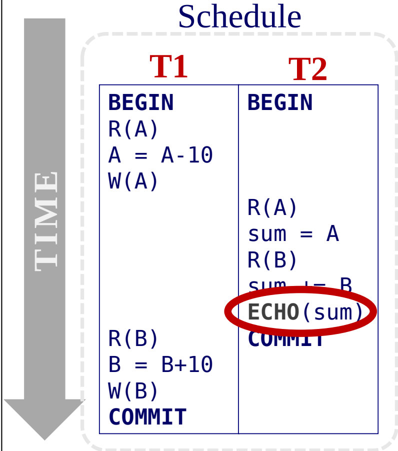


Example #3 – Threesome

- **Q:** Is this equivalent to a serial execution?
- **A:** Yes (T2, T1, T3)
  - Notice that T3 should go after T2, although it starts before it!

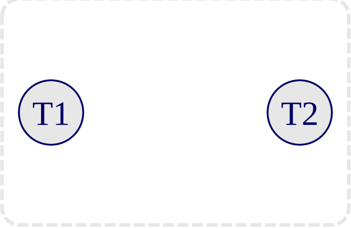
Example #4 – Inconsistent Analysis

Schedule



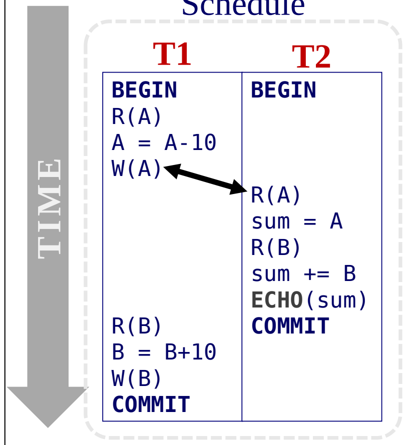
T1	T2
BEGIN	**BEGIN**
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO(sum)
R(B)	**COMMIT**
B = B+10	
W(B)	
COMMIT	

Dependency Graph



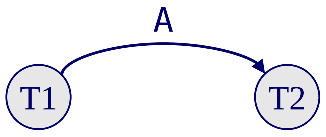
Example #4 – Inconsistent Analysis

Schedule



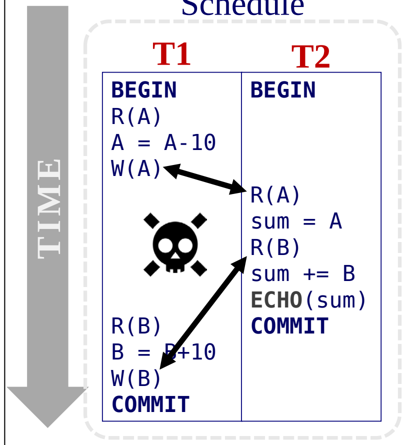
T1	T2
BEGIN	**BEGIN**
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO(sum)
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph



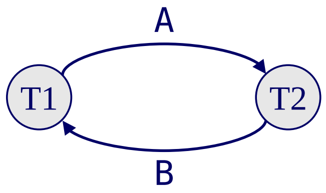
Example #4 – Inconsistent Analysis

Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
A = A-10	
W(A)	
	R(A)
	sum = A
	R(B)
	sum += B
	ECHO(sum)
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

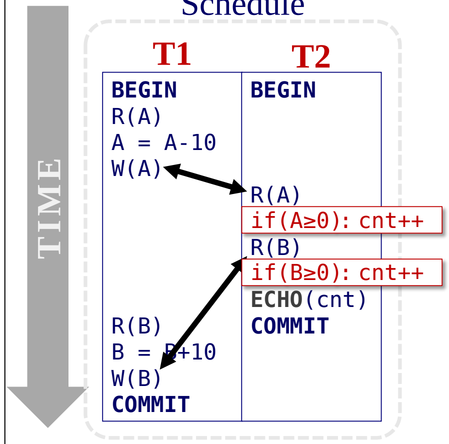
Dependency Graph



Is it possible to create a schedule similar to this that is “correct” but still not conflict serializable?

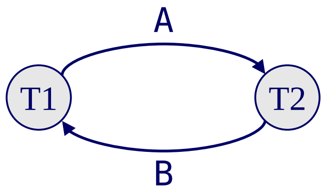
Example #4 – Inconsistent Analysis

Schedule



T1	T2
BEGIN	**BEGIN**
R(A)	
A = A-10	
W(A)	
	R(A)
	if(A≥0): cnt++
	R(B)
	if(B≥0): cnt++
	ECHO(cnt)
	COMMIT
R(B)	
B = B+10	
W(B)	
COMMIT	

Dependency Graph

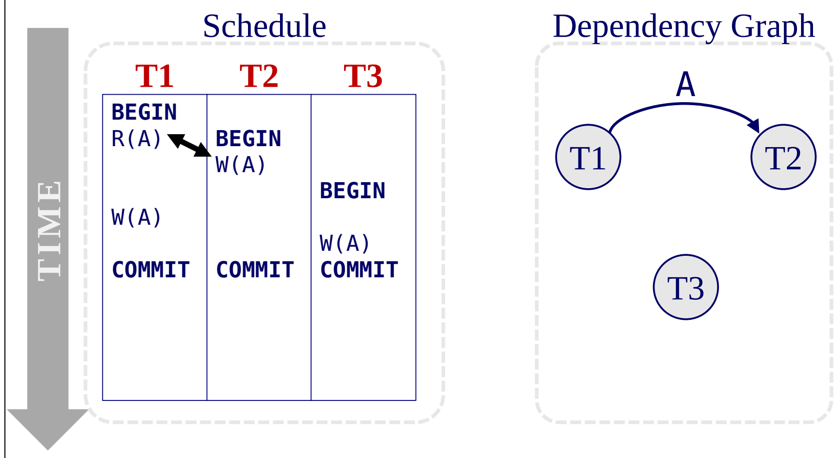


T2 counts the number of active accounts.

View Serializability

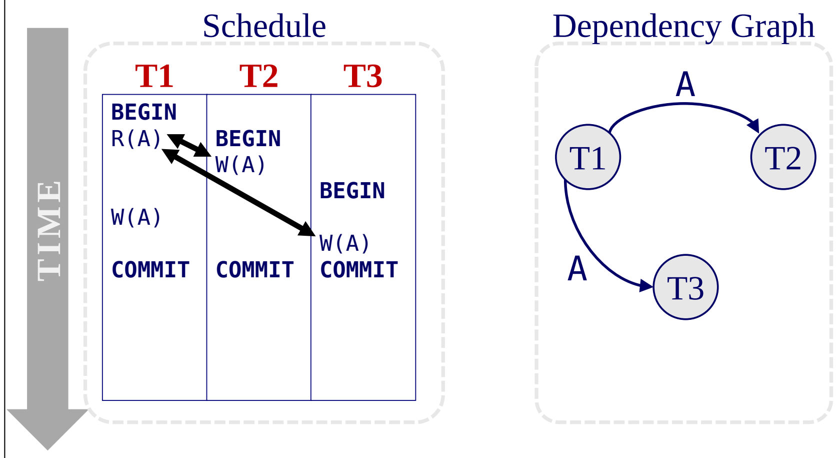
- Alternative (weaker) notion of serializability.
- Schedules S1 and S2 are ***view equivalent*** if:
  - If T1 reads initial value of A in S1, then T1 also reads initial value of A in S2.
  - If T1 reads value of A written by T2 in S1, then T1 also reads value of A written by T2 in S2.
  - If T1 writes final value of A in S1, then T1 also writes final value of A in S2.

View Serializability



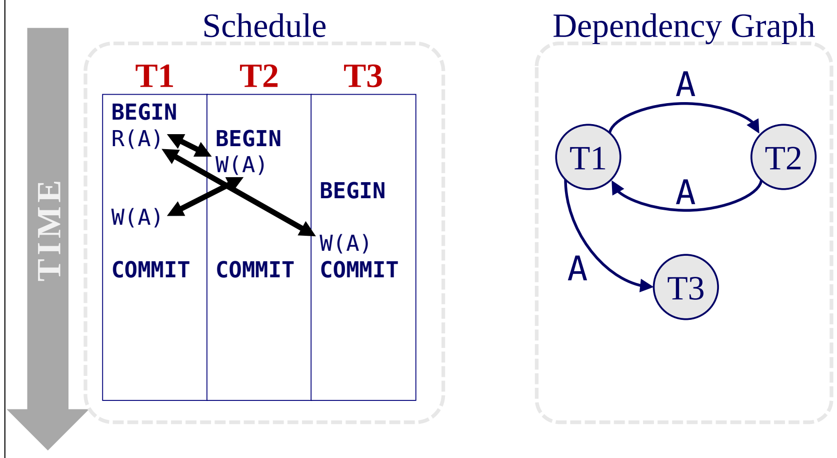
T1	T2	T3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

View Serializability



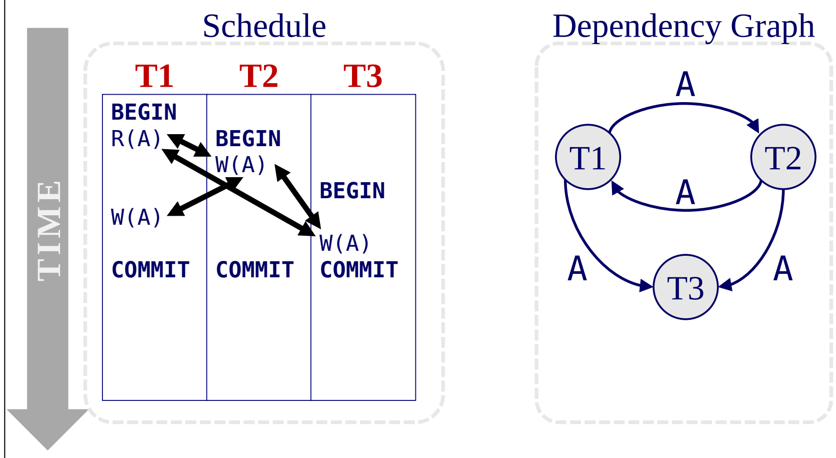
T1	T2	T3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

View Serializability



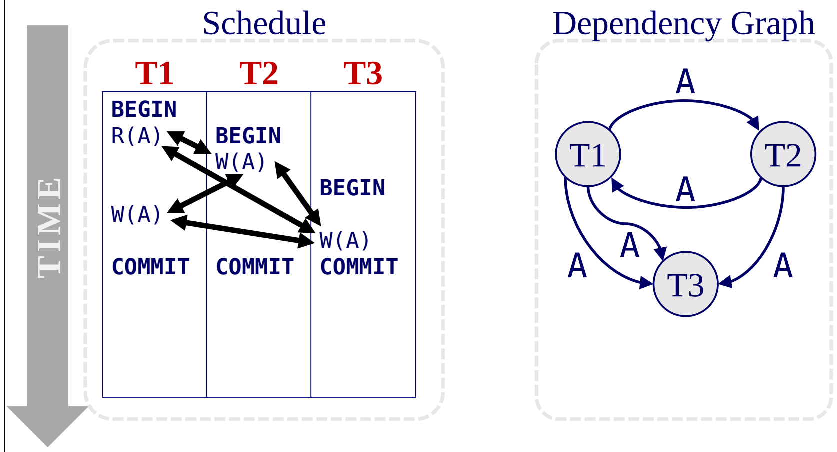
T1	T2	T3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

View Serializability



T1	T2	T3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

View Serializability

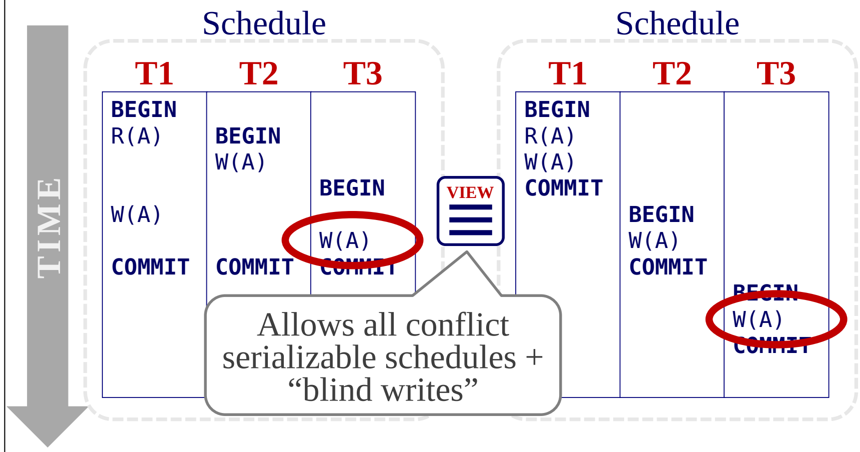


Schedule

T1	T2	T3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Dependency Graph

View Serializability



Schedule

T1	T2	T3
BEGIN		
R(A)	BEGIN	
	W(A)	
		BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT

Schedule

T1	T2	T3
BEGIN		
R(A)		
W(A)		
COMMIT		
	BEGIN	
	W(A)	
	COMMIT	
		BEGIN
		W(A)
		COMMIT

Allows all conflict serializable schedules + “blind writes”

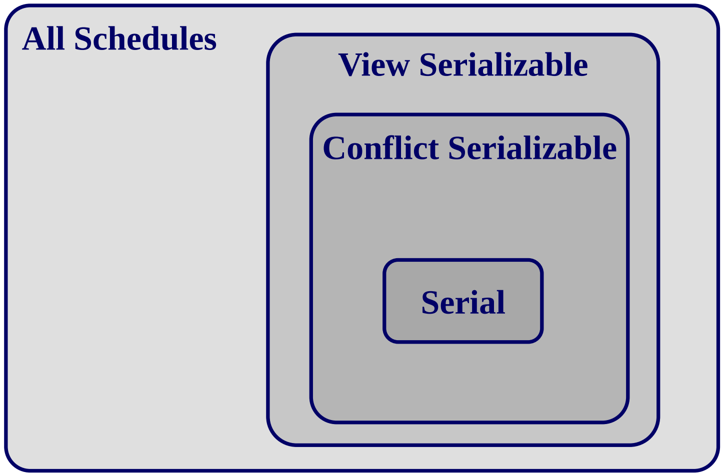
Serializability

- **View Serializability** allows (slightly) more schedules than **Conflict Serializability** does.
  - But is difficult to enforce efficiently.
- Neither definition allows all schedules that you would consider “serializable”.
  - This is because they don’t understand the meanings of the operations or the data (recall example #4)

Serializability

- In practice, **Conflict Serializability** is what gets used, because it can be enforced efficiently.
- To allow more concurrency, some special cases get handled separately at the application level.

Schedules



All Schedules

View Serializable

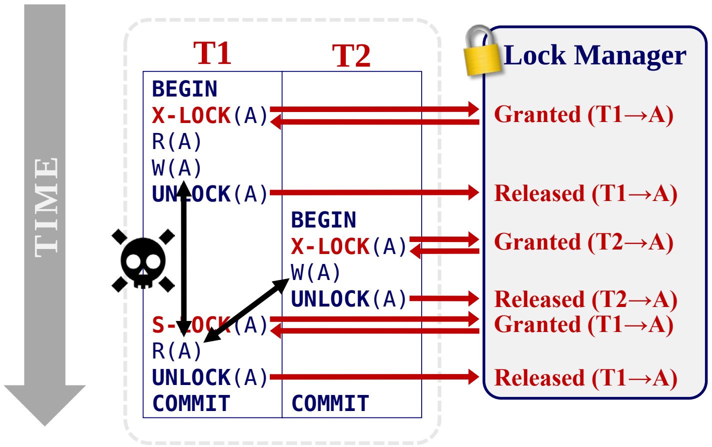
Conflict Serializable

Serial

Today’s Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities

Executing with Locks



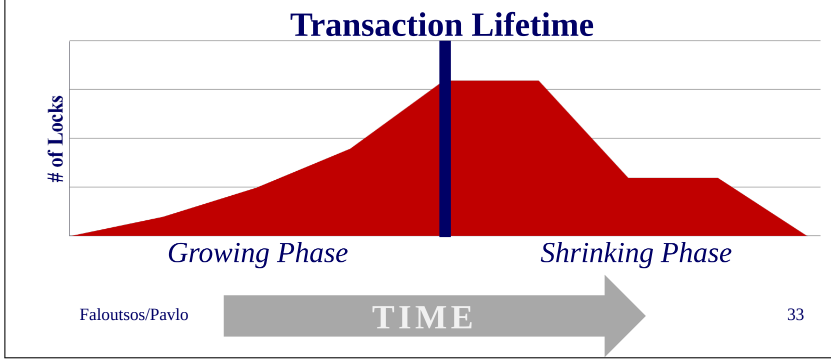
T1	T2	Lock Manager
BEGIN		
X-LOCK(A)		Granted (T1→A)
R(A)		
W(A)		
UNLOCK(A)		Released (T1→A)
	BEGIN	
	X-LOCK(A)	Granted (T2→A)
	W(A)	
	UNLOCK(A)	Released (T2→A)
S-LOCK(A)		Granted (T1→A)
R(A)		
UNLOCK(A)		Released (T1→A)
COMMIT	COMMIT	

Two-Phase Locking

- **Phase 1: Growing**
  - Each txn requests the locks that it needs from the DBMS’s lock manager.
  - The lock manager grants/denies lock requests.
- **Phase 2: Shrinking**
  - The txn is allowed to only release locks that it previously acquired. It cannot acquire new locks.

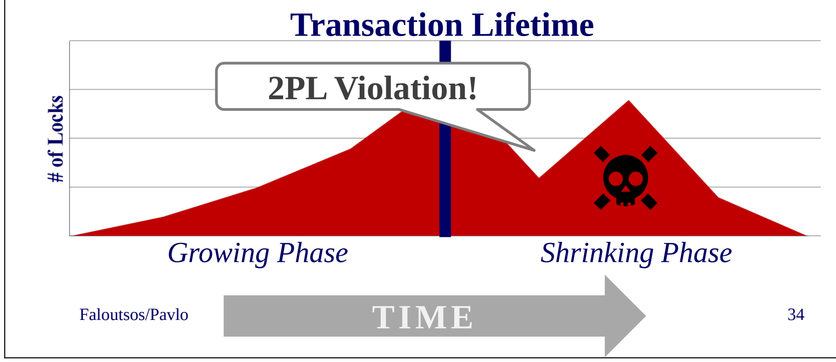
Two-Phase Locking

- The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

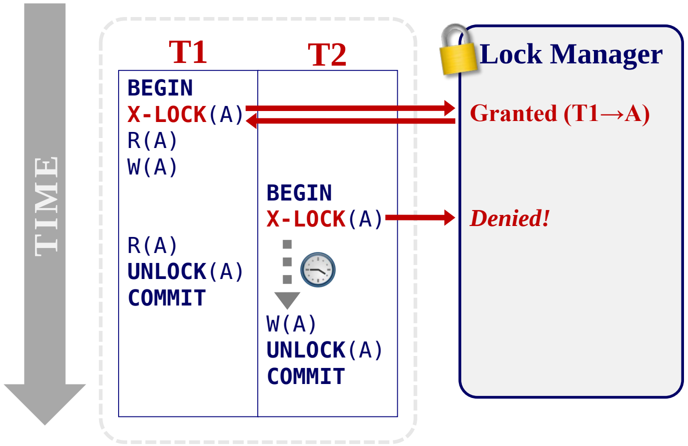


Two-Phase Locking

- The txn is not allowed to acquire/upgrade locks after the growing phase finishes.

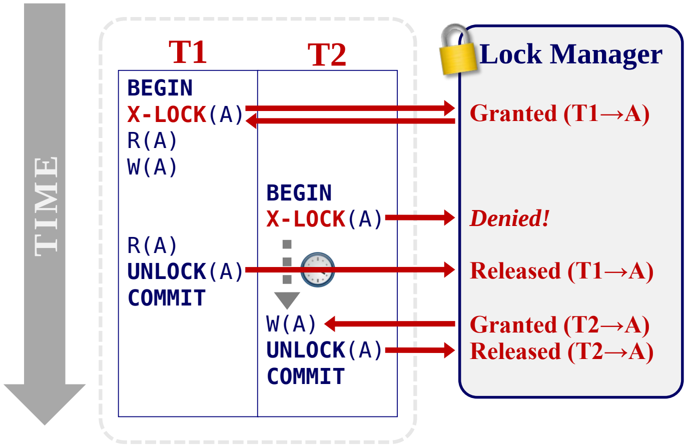


Executing with 2PL



T1	T2	Lock Manager
BEGIN		
X-LOCK(A)		**Granted (T1→A)**
R(A)		
W(A)		
	BEGIN	
	X-LOCK(A)	***Denied!***
R(A)		
UNLOCK(A)		
COMMIT		
	W(A)	
	UNLOCK(A)	
	COMMIT	

Executing with 2PL

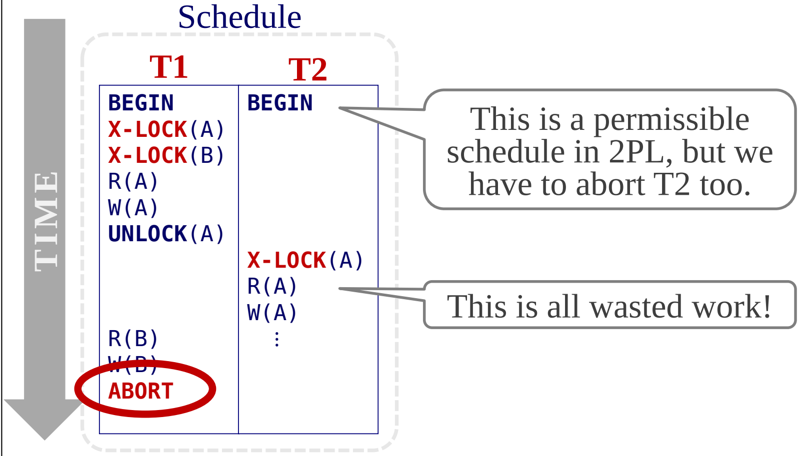


T1	T2	Lock Manager
BEGIN		
X-LOCK(A)		**Granted (T1→A)**
R(A)		
W(A)		
	BEGIN	
	X-LOCK(A)	***Denied!***
R(A)		
UNLOCK(A)		**Released (T1→A)**
COMMIT		
	W(A)	**Granted (T2→A)**
	UNLOCK(A)	**Released (T2→A)**
	COMMIT	

Two-Phase Locking

- 2PL on its own is sufficient to guarantee conflict serializability (i.e., schedules whose precedence graph is acyclic), but, it is subject to *cascading aborts*.

2PL – Cascading Aborts



Schedule

T1	T2
BEGIN	BEGIN
X-LOCK(A)	
X-LOCK(B)	
R(A)	
W(A)	
UNLOCK(A)	
	X-LOCK(A)
	R(A)
	W(A)
R(B)	⋮
W(B)	
ABORT	

This is a permissible schedule in 2PL, but we have to abort T2 too.

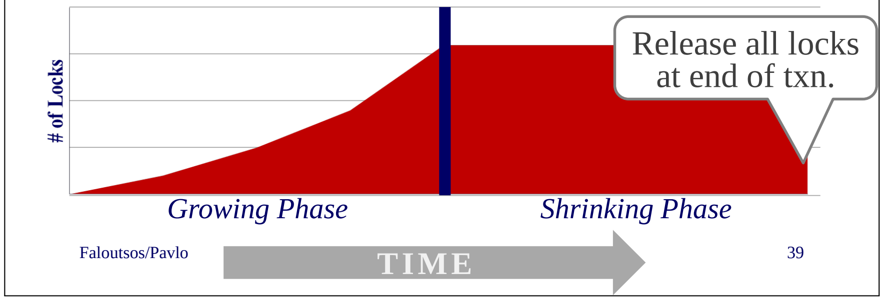
This is all wasted work!

2PL Observations

- There are schedules that are serializable but would not be allowed by 2PL.
- Locking limits concurrency.
- May lead to deadlocks.
- May still have “dirty reads”
  - Solution: **Strict 2PL**

Strict Two-Phase Locking

- The txn is not allowed to acquire/upgrade locks after the growing phase finishes.
- Allows only conflict serializable schedules, but it is actually stronger than needed.



Release all locks at end of txn.

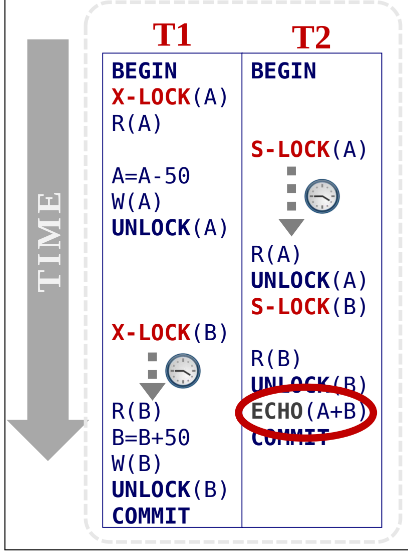
Strict Two-Phase Locking

- A schedule is ***strict*** if a value written by a txn is not read or overwritten by other txns until that txn finishes.
- Advantages:
  - Does not incur cascading aborts.
  - Aborted txns can be undone by just restoring original values of modified tuples.

Examples

- **T1:** *Move $50 from Andy's account to his bookie's account.*
- **T2:** *Compute the total amount in all accounts and return it to the application.*
- Legend:
  - **A** → Andy's account.
  - **B** → The bookie's account.

Non-2PL Example



T1	T2
BEGIN	**BEGIN**
X-LOCK(A)	
R(A)	
	S-LOCK(A)
A=A-50	
W(A)	
UNLOCK(A)	
	R(A)
	UNLOCK(A)
	S-LOCK(B)
X-LOCK(B)	
	R(B)
	UNLOCK(B)
R(B)	**ECHO**(A+B)
B=B+50	**COMMIT**
W(B)	
UNLOCK(B)	
COMMIT	

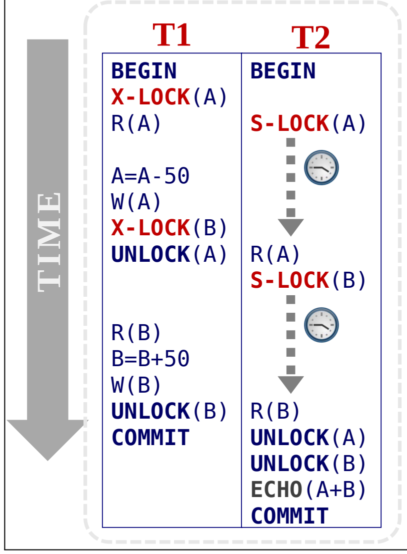
Initial State

A=100, B=100

T2 Output

150

2PL Example



T1	T2
BEGIN	**BEGIN**
X-LOCK(A)	
R(A)	**S-LOCK**(A)
A=A-50	
W(A)	
X-LOCK(B)	
UNLOCK(A)	R(A)
	S-LOCK(B)
R(B)	
B=B+50	
W(B)	
UNLOCK(B)	R(B)
COMMIT	**UNLOCK**(A)
	UNLOCK(B)
	ECHO(A+B)
	COMMIT

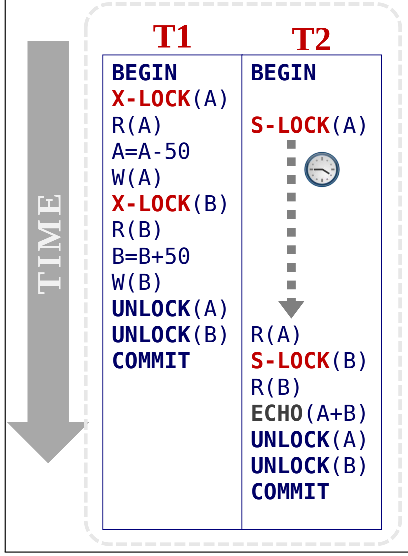
Initial State

A=100, B=100

T2 Output

200

Strict 2PL Example



T1	T2
BEGIN	BEGIN
X-LOCK(A)	
R(A)	S-LOCK(A)
A=A-50	
W(A)	
X-LOCK(B)	
R(B)	
B=B+50	
W(B)	
UNLOCK(A)	
UNLOCK(B)	R(A)
COMMIT	S-LOCK(B)
	R(B)
	ECHO(A+B)
	UNLOCK(A)
	UNLOCK(B)
	COMMIT

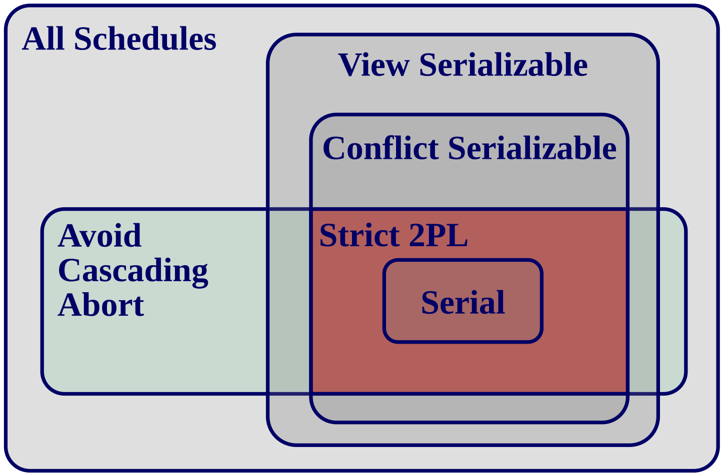
Initial State

A=100, B=100

T2 Output

200

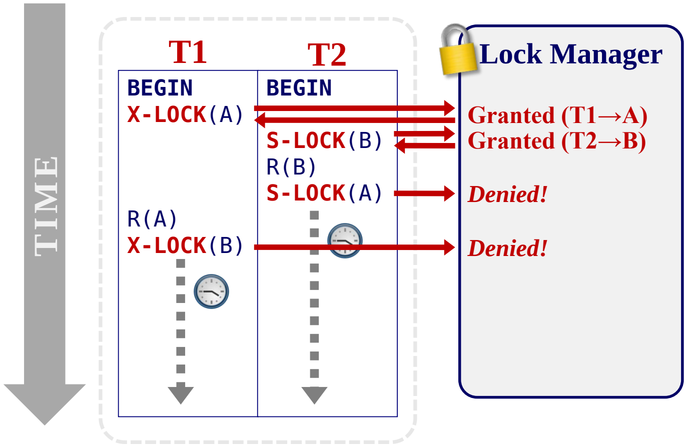
Schedules



Today’s Class

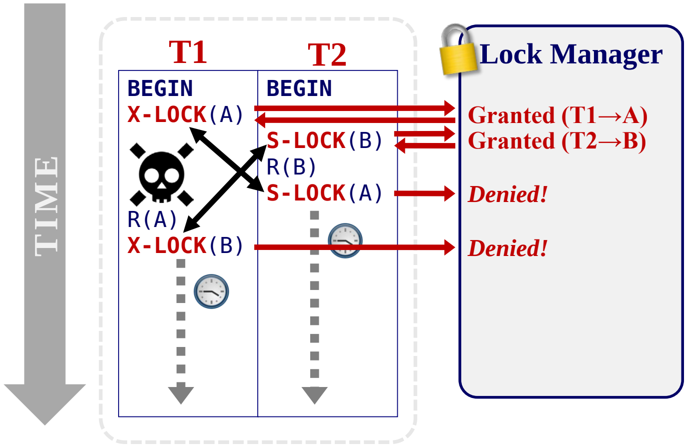
- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities

It Just Got Real, Son



T1	T2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

It Just Got Real, Son



T1	T2
BEGIN	BEGIN
X-LOCK(A)	
	S-LOCK(B)
	R(B)
	S-LOCK(A)
R(A)	
X-LOCK(B)	

Lock Manager

Granted (T1→A)

Granted (T2→B)

Denied!

Denied!

Deadlocks

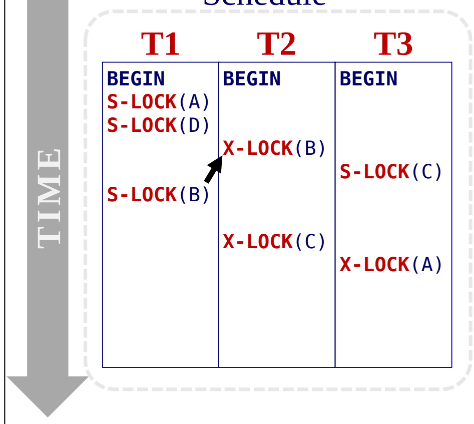
- **Deadlock:** Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock detection
  - Deadlock prevention

Deadlock Detection

- The DBMS creates a ***waits-for*** graph:
  - Nodes are transactions
  - Edge from T_i to T_j if T_i is waiting for T_j to release a lock
- The system periodically check for cycles in ***waits-for*** graph.

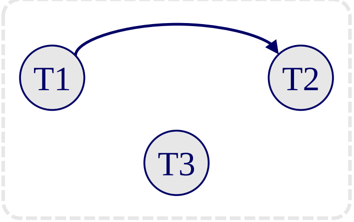
Deadlock Detection

Schedule

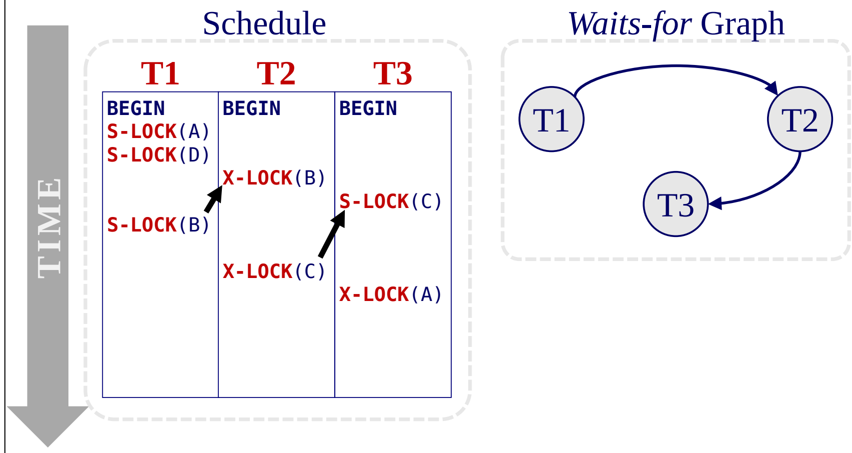


T1	T2	T3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
S-LOCK(D)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-for Graph



Deadlock Detection

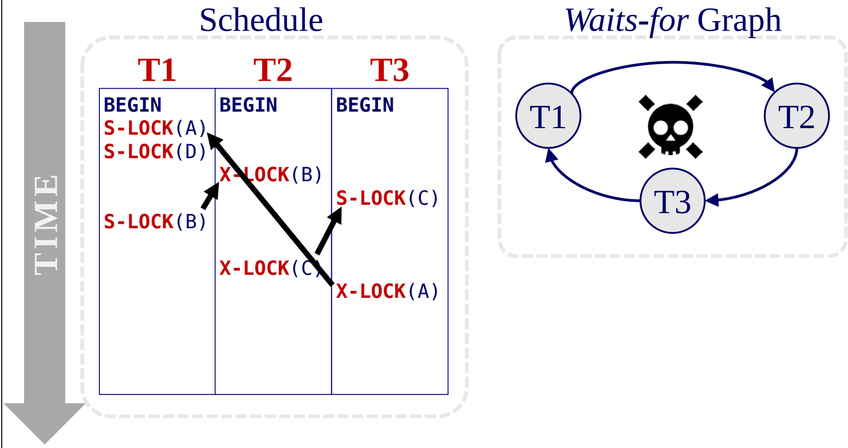


Schedule

T1	T2	T3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
S-LOCK(D)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-for Graph

Deadlock Detection



Schedule

T1	T2	T3
BEGIN	BEGIN	BEGIN
S-LOCK(A)		
S-LOCK(D)		
	X-LOCK(B)	
		S-LOCK(C)
S-LOCK(B)		
	X-LOCK(C)	
		X-LOCK(A)

Waits-for Graph

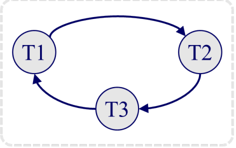
Deadlock Detection

- How often should we run the algorithm?
- How many txns are typically involved?
- What do we do when we find a deadlock?

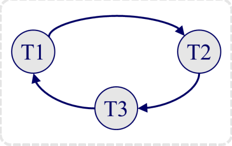
Deadlock Handling

- **Q:** What do we do?
- **A:** Select a "victim" and rollback it back to break the deadlock.

Waits-for Graph



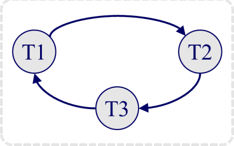
Deadlock Handling



Waits-for Graph

- **Q:** Which one do we choose?
- **A:** It depends…
  - By age (lowest timestamp)
  - By progress (least/most queries executed)
  - By the # of items already locked
  - By the # of txns that we have to rollback with it
- We also should consider the # of times a txn has been restarted in the past.

Deadlock Handling



Waits-for Graph

- **Q:** How far do we rollback?
- **A:** It depends…
  - Completely
  - Minimally (i.e., just enough to release locks)

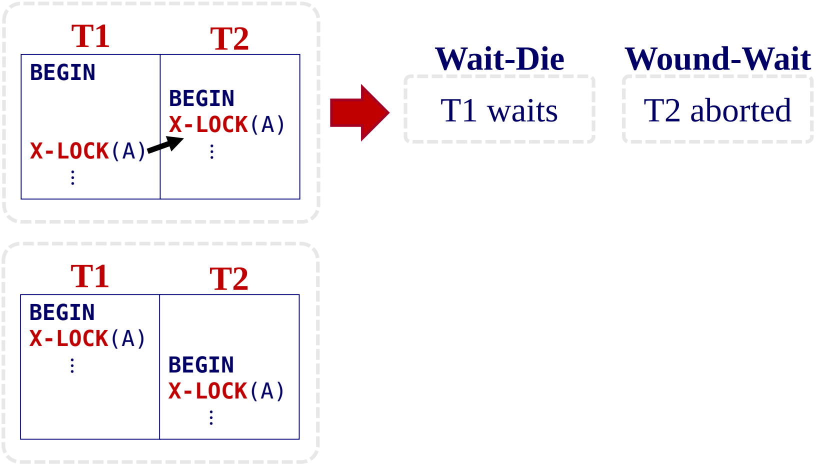
Deadlock Prevention

- When a txn tries to acquire a lock that is held by another txn, kill one of them to prevent a deadlock.
- No *waits-for* graph or detection algorithm.

Deadlock Prevention

- Assign priorities based on timestamps:
  - Older → higher priority (e.g., T1 > T2)
- Two different prevention policies:
  - **Wait-Die:** If T1 has higher priority, T1 waits for T2; otherwise T1 aborts (“old wait for young”)
  - **Wound-Wait:** If T1 has higher priority, T2 aborts; otherwise T1 waits (“young wait for old”)

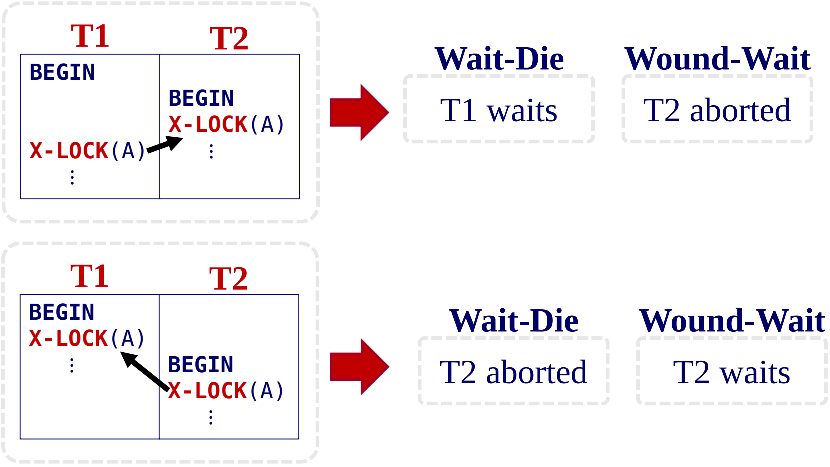
Deadlock Prevention



Wait-Die
T1 waits

Wound-Wait
T2 aborted

Deadlock Prevention



Wait-Die
T1 waits

Wound-Wait
T2 aborted

Wait-Die
T2 aborted

Wound-Wait
T2 waits

Deadlock Prevention

- **Q:** Why do these schemes guarantee no deadlocks?
- **A:** Only one “type” of direction allowed.

- **Q:** When a transaction restarts, what is its (new) priority?
- **A:** Its original timestamp. Why?

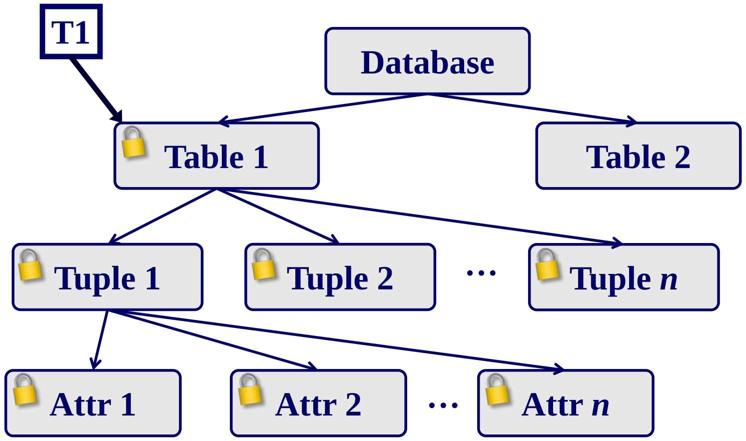
Today’s Class

- Serializability
- Two-Phase Locking
- Deadlocks
- Lock Granularities

Lock Granularities

- When we say that a txn acquires a “lock”, what does that actually mean?
  - On an Attribute? Tuple? Page? Table?
- Ideally, each txn should obtain fewest number of locks that is needed…

Database Lock Hierarchy



T1

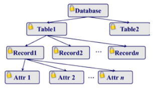
Database

Table 1 **Table 2**

Tuple 1 **Tuple 2** … **Tuple *n***

Attr 1 **Attr 2** … **Attr *n***

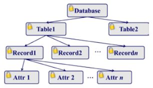
Example



- **T1:** Get the balance of Andy’s shady off-shore bank account.
- **T2:** Increase Christos’ bank account balance by 1%.

- **Q:** What locks should they obtain?

Example



- **Q:** What locks should they obtain?
- **A:** Multiple
  - **Exclusive** + **Shared** for leafs of lock tree.
  - Special **Intention** locks for higher levels

Intention Locks

- Intention locks allow a higher level node to be locked in **S** or **X** mode without having to check all descendent nodes.
- If a node is in an intention mode, then explicit locking is being done at a lower level in the tree.

Intention Locks

- **Intention-Shared** (**IS**): Indicates explicit locking at a lower level with shared locks.
- **Intention-Exclusive** (**IX**): Indicates locking at lower level with exclusive or shared locks.

Intention Locks

- **Shared+Intention-Exclusive** (**SIX**): The subtree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

Compatibility Matrix

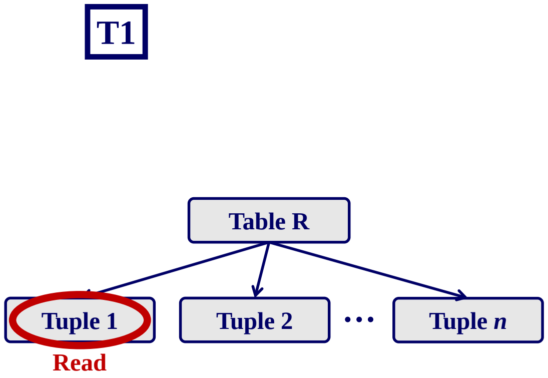
T2 Wants

T1 Holds

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

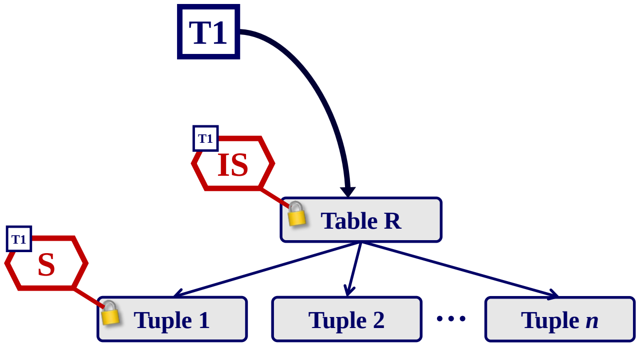
Example – Two-level Hierarchy

Read Andy’s record in **R**.



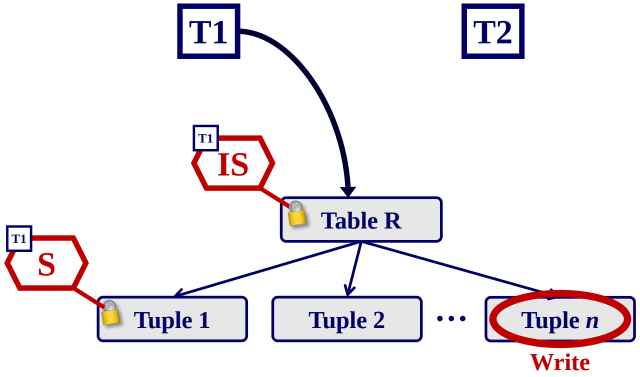
Example – Two-level Hierarchy

Read Andy’s record in **R**.



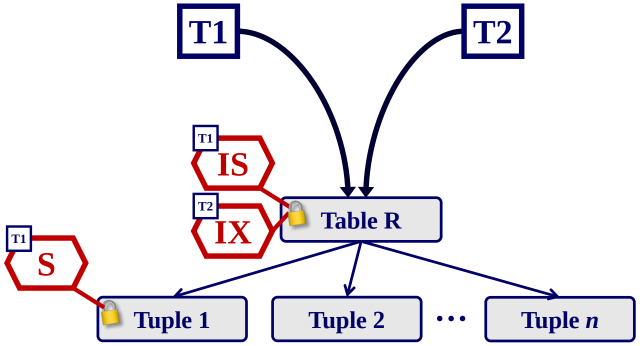
Example – Two-level Hierarchy

Update Christos’ record in **R**.



Example – Two-level Hierarchy

Update Christos’ record in **R**.



Example – Two-level Hierarchy

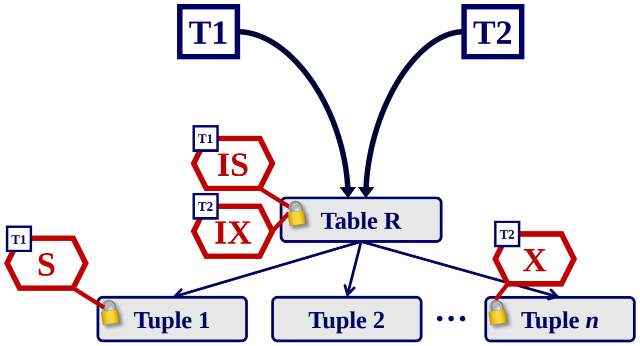
in **R**.

T2 Wants

T1 Holds	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
SIX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

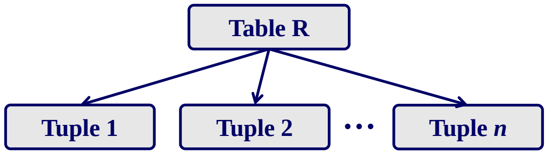
Example – Two-level Hierarchy

Update Christos' record in **R**.



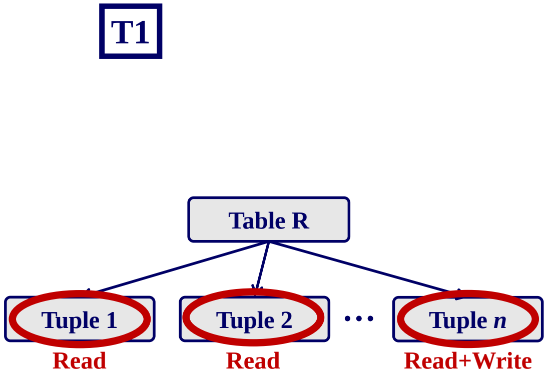
Example – Threesome

- Assume three txns execute at same time:
  - **T1:** Scan **R** and update a few tuples.
  - **T2:** Read a single tuple in **R**.
  - **T3:** Scan all tuples in **R**.



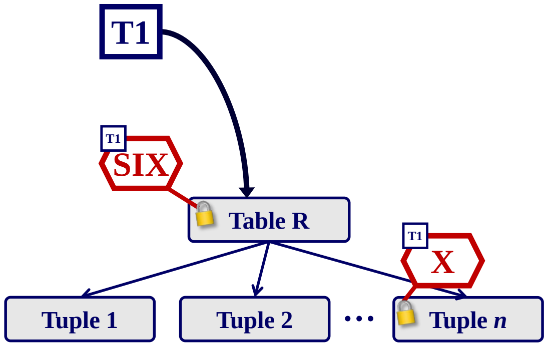
Example – Threesome

Scan **R** and update a few tuples.



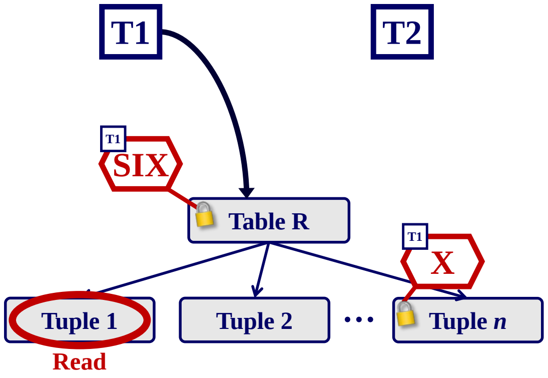
Example – Threesome

Scan **R** and update a few tuples.



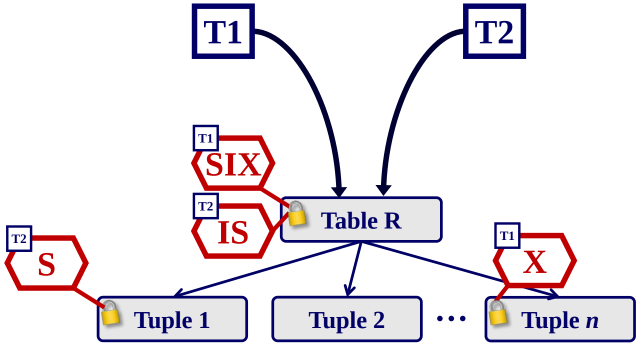
Example – Threesome

Read a single tuple in **R**.



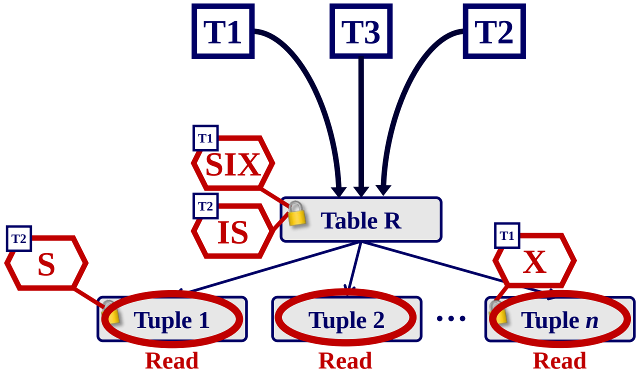
Example – Threesome

Read a single tuple in **R**.



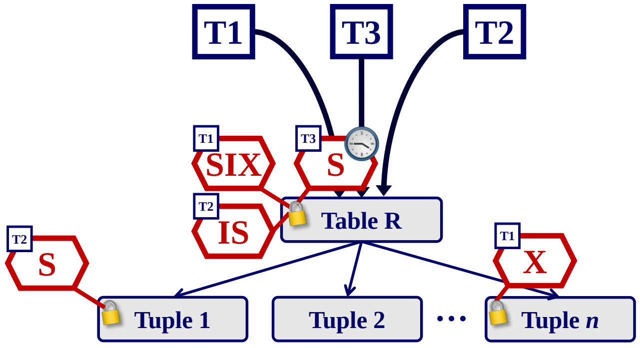
Example – Threesome

Scan all tuples in **R**.



Example – Threesome

Scan all tuples in **R**.



Multiple Lock Granularities

- Useful in practice as each txn only needs a few locks.
- Intention locks help improve concurrency:
  - **Intention-Shared** (**IS**): Intent to get **S** lock(s) at finer granularity.
  - **Intention-Exclusive (IX):** Intent to get **X** lock(s) at finer granularity.
  - **Shared+Intention-Exclusive** (**SIX**): Like **S** and **IX** at the same time.

Locking in Practice

- You typically don’t set locks manually.
- Sometimes you will need to provide the DBMS with hints to help it to improve concurrency.
- Also useful for doing major changes.

LOCK TABLE

Postgres

```
LOCK TABLE <table> IN <mode> MODE;
```

MySQL

```
LOCK TABLE <table> <mode>;
```

- Explicitly locks a table.
- Not part of the SQL standard.
  - Postgres Modes: `SHARE`, `EXCLUSIVE`
  - MySQL Modes: `READ`, `WRITE`

SELECT...FOR UPDATE

```
SELECT * FROM <table>
 WHERE <qualification> FOR UPDATE;
```

- Perform a select and then sets an exclusive lock on the matching tuples.
- Can also set shared locks:
  - Postgres: `FOR SHARE`
  - MySQL: `LOCK IN SHARE MODE`

Concurrency Control Summary

- Conflict Serializability ↔ Correctness
- Automatically correct interleavings:
  - Locks + protocol (2PL, S2PL ...)
  - Deadlock detection + handling
  - Deadlock prevention