CSC553 Advanced Database Concepts

Tanu Malik

School of Computing

DePaul University

- So far we have seen sufficient conditions that allows us to check whether the schedule is serializable.
    - Serial
    - Serializable
    - Conflict serializable
    - View serializable
    - Recoverability
        - Recoverable
        - Avoids cascading deletes

Example1

- T_1 R(A) W(A) R(B) W(B)
- T_2 R(A) W(A) R(B) W(B)

A Non-serializable Schedule

- T_1 R(A) W(A) R(B) W(B)
- T_2 R(A) W(A) R(B) W(B)

- But how to ensure serializability during **runtime**?

- Challenge: The system does not know in advance which transactions will run and which items they will access.

Scheduler

- The scheduler:
  - Module that schedules the transaction’s actions, ensuring serializability

- Two main approaches
  - Pessimistic:
    - lock-based concurrency control,
    - Timestamp-based concurrency control,
    - needs deadlock detection
  - Optimistic: tracking of read-set/write-set, validation before commit.
  - Multi-version: less concurrency overhead for read-only queries

Pessimistic Scheduler

- Simple idea:
  - Each database object has a **unique** lock
  - Each transaction must first **acquire** the lock before reading/writing that DB object
  - If the lock is taken by another transaction, then wait
  - The transaction must **release** the lock(s)

Notation

- L_i (A) = transaction T_i **acquires** lock for element A
- U_i (A) = transaction T_i **releases** lock for element A

- A lock is a small bookkeeping object associated with a database object.

A Non-serializable Schedule

- T_1 R(A) W(A) R(B) W(B)
- T_2 R(A) W(A) R(B) W(B)

A Non-serializable Schedule

- T_1: $L_1(A)$ R(A) W(A) $U_1(A)$ $L_1(B)$
- T_2: $L_2(A)$R(A) W(A) $U_2(A)$ $L_2(B)$ Denied…

t

- T_1: R(B) W(B) $U_1(B)$
- T_2: Granted…R(B) W(B)$U_2(B)$

Locks ensure a conflict-serializable schedule

- T_1: $L_1(A)$ R(A) W(A) $U_1(A)$ $L_1(B)$
- T_2: $L_2(A)$R(A) W(A) $U_2(A)$ $L_2(B)$ Denied…

t

- T_1: R(B) W(B) $U_1(B)$
- T_2: Granted…R(B) W(B)$U_2(B)$

Example2

- T_1: $L_1(A)$ R(A) W(A) $U_1(A)$
- T_2: $L_2(A)$R(A) W(A) $U_2(A)$ $L_2(B)$ R(B)

t

- T_1: $L_1(B)$ R(B) W(B) $U_1(B)$
- T_2: W(B) $U_2(B)$

Locks did not enforce conflict serializability!!! What's wrong?

Two-Phase Locking

- The 2PL protocol:
  - In every transaction, all lock requests must precede all unlock requests
  - This ensures conflict serializability! (will prove this shortly)

- Note: A protocol is a set of rules to be followed by each txn such that interleaved actions of a transaction are conflict-serializable.

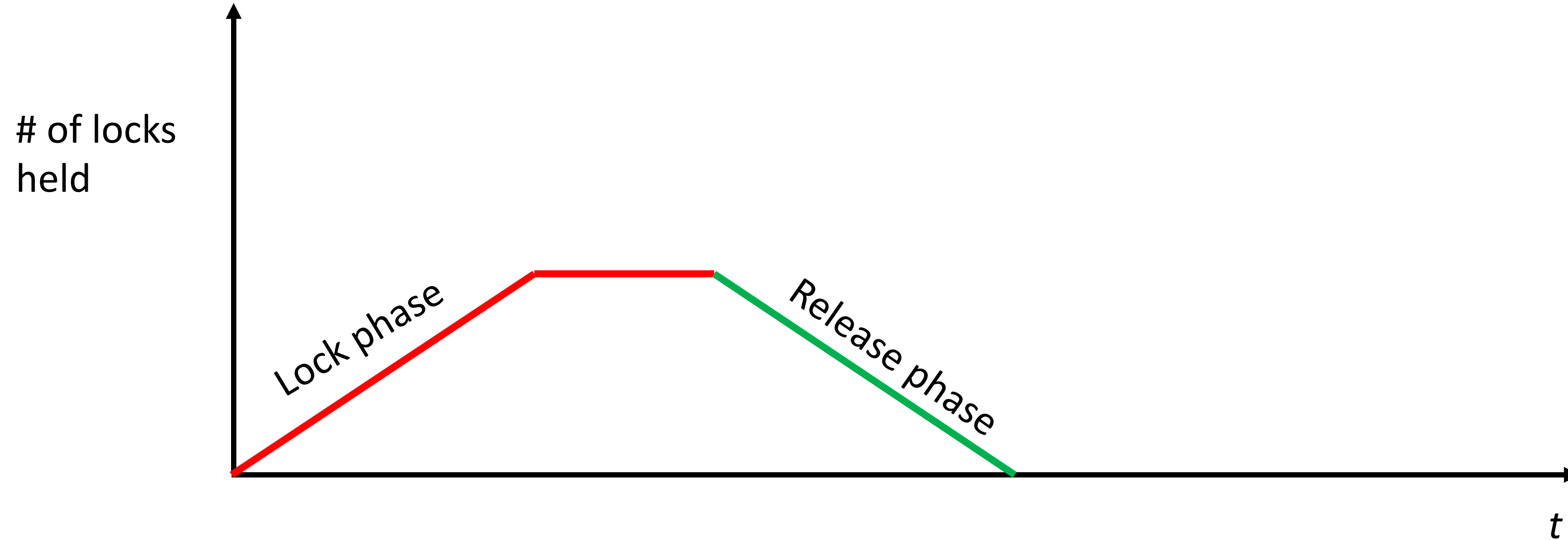
Conflict-serializable with 2PL

- T_1: **L_1(A) L_1(B)** R(A) W(A) U_1(A)
- T_2: L_2(A)R(A) W(A) L_2(B) Denied

t

- T_1: R(B) W(B) U_1(B)
- T_2: Granted R(B) W(B) U_2(A) U_2(B)

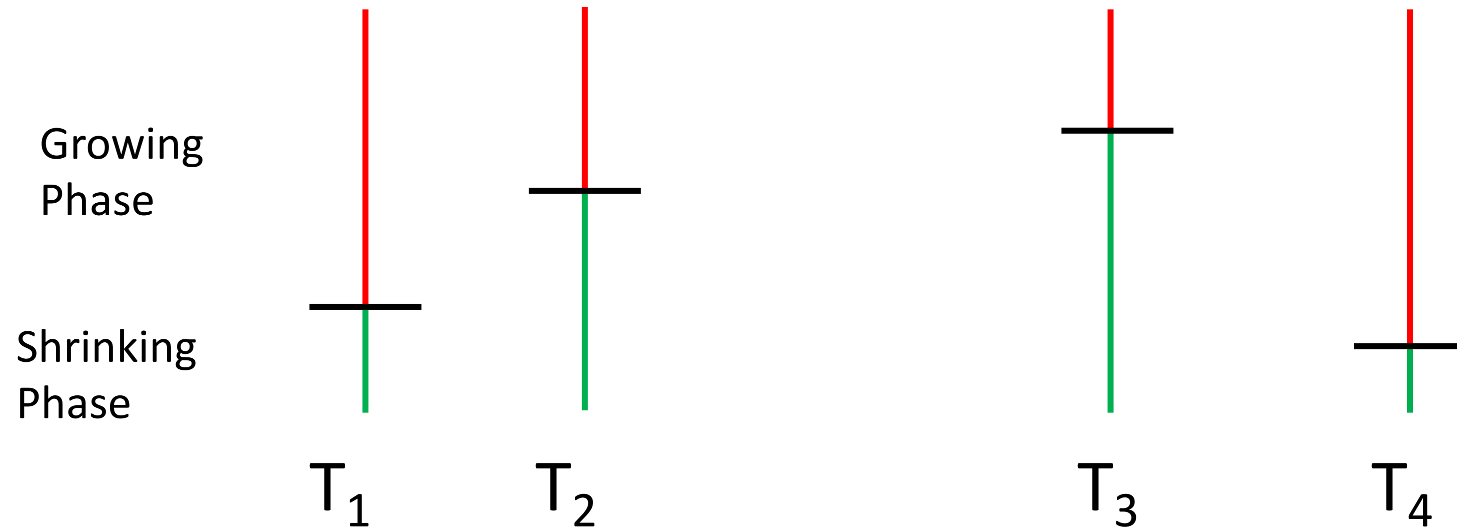
2PL Growing and Shrinking



2PL Growing and Shrinking

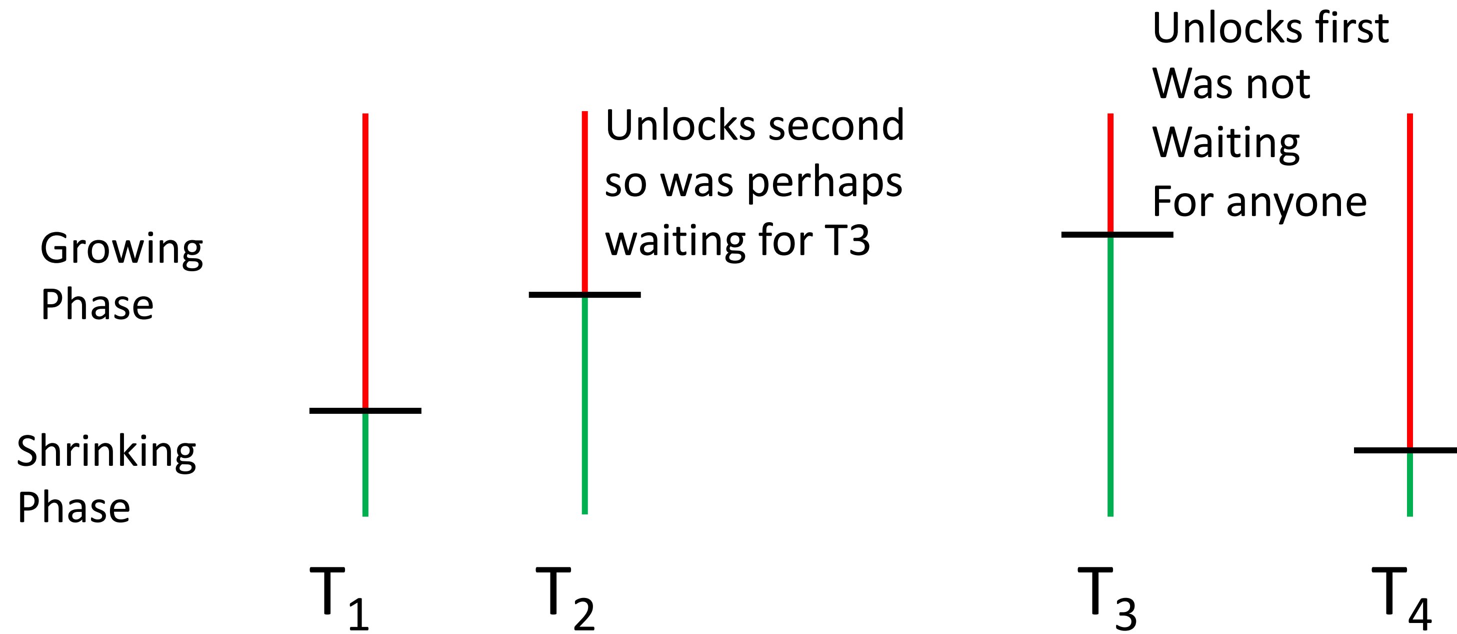
- Protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks

Multiple Transactions



- Equivalent to each transaction executing entirely the moment it enters shrinking phase

Multiple Transactions



- Equivalent to each transaction executing entirely the moment it enters shrinking phase

Are these schedules in 2PL?

- T_1: L(B) W(B) U(B)
- T_2: L(A) R(A) U(A) L(B) W(B) U(B)
- T_1: L(B) W(B) U(B)
- T_2: L(A) L(B) R(A) W(B) U(A) U(B)
- T_1: L(B) W(B) U(B)
- T_2: L(A) R(A) U(A) L(B) W(B) U(B)
- T_1: L(B) W(B) U(B)
- T_2: L(A) R(A) L(B) U(A) W(B) U(B)

2PL Growing and Shrinking

- Protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

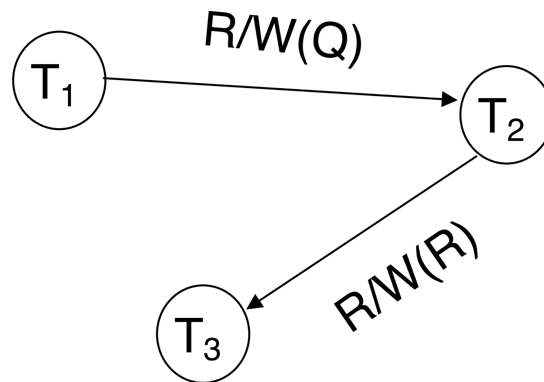
Theorem

- 2PL ensures conflict serializability

2PL & Serializability

- Recall: Precedence Graph

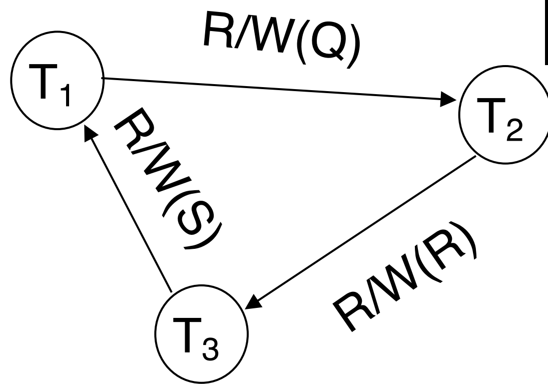
T_1	T_2	T_3
read(Q)		
	write(Q)	
	read(R)	
		write(R)
		read(S)



2PL & Serializability

- Recall: Precedence Graph

T_1	T_2	T_3
read(Q)		
	write(Q)	
	read(R)	
		write(R)
		read(S)
write(S)		



Cycle → Non-serializable

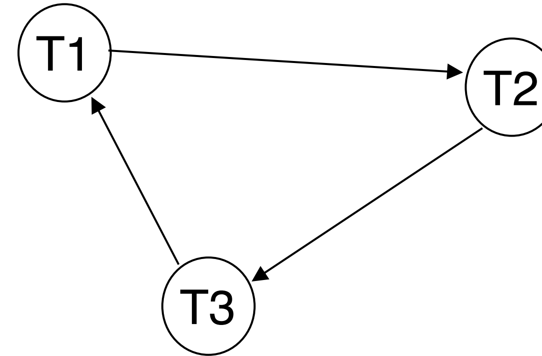
2PL & Serializability

Relation between Growing & Shrinking phase:

$T_1G < T_1S$

$T_2G < T_2S$

$T_3G < T_3S$



T1 must release locks for other to proceed

$T_1S < T_2G$

$T_2S < T_3G$

$T_3S < T_1G$

$T_1G < T_1S < T_2G < T_2S < T_3G < T_3S < T_1G$

Not Possible under 2PL!

It can be generalized for any set of transactions...

The Locking Scheduler

- Task 1: act on behalf of the transaction

- Add lock/unlock requests to transactions
- Examine all READ(A) or WRITE(A) actions
- Add appropriate lock requests
- On COMMIT/ROLLBACK release all locks
- Ensures Strict 2PL!

The Locking Scheduler

- Task 2: -- act on behalf of the system

Execute the locks accordingly

- Lock table: a big, critical data structure in a DBMS !
- When a lock is requested, check the lock table
- Grant, or add the transaction to the database object's wait list
- When a lock is released, re-activate a transaction from its wait list
- When a transaction aborts, release all its locks
- Check for deadlocks occasionally

Lock Granularity

- Granularity of locking is a tradeoff

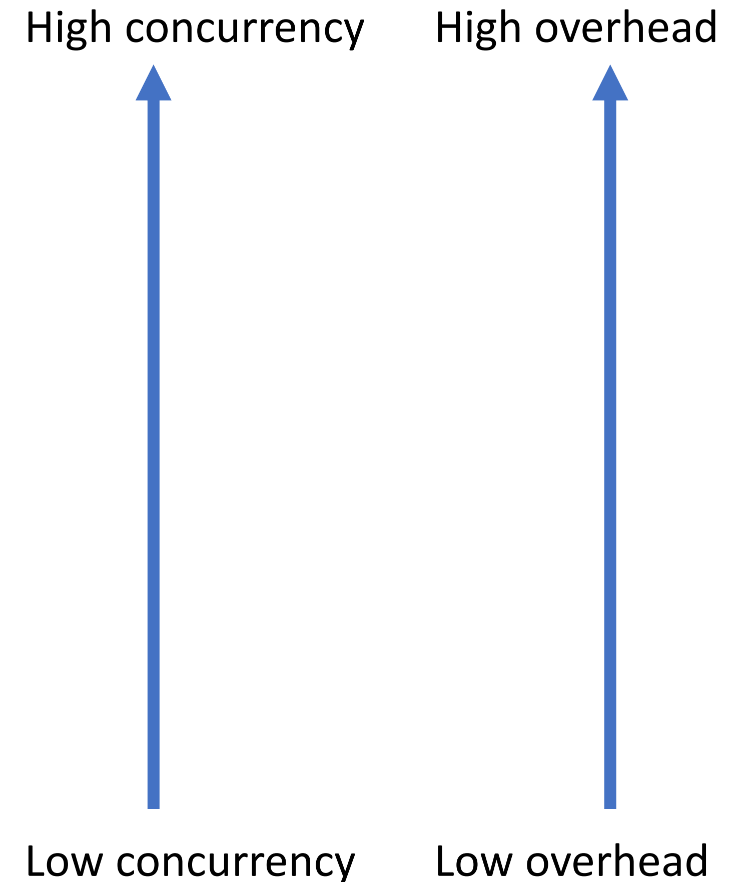
- Fine granularity locking (e.g., tuples)

- Coarse grain locking (e.g., tables, predicate locks)

- Very-coarse grained level (database level)

Lock Granularity

- Granularity of locking is a tradeoff
- Fine granularity locking (e.g., tuples)
- Coarse grain locking (e.g., tables, predicate locks)
- Very-coarse grained level (database level)



Multi-granularity Locking

- S = shared lock (for READ)
  - Acquired before reading a database object
  - Many transactions can hold on to a shared lock at the same time.
- X = exclusive lock (for WRITE)
  - Exclusive lock is acquired on a database object before writing the object (in memory).
  - A transaction can hold onto an exclusive lock on a database object only if no other transaction holds onto an exclusive lock on the same object.
  - If a transaction holds an exclusive lock on a database object it can also READ the object.

X lock is stronger than a S lock

Multi-granularity Locking

- S = shared lock (for READ)
- X = exclusive lock (for WRITE)

- Lock Compatibility Matrix

	None	S	X
None			
S			
X			

Example of Schedule with S and X locks

- T_1 X(B) W(B) U(B)
- T_2 S(A) R(A) X(B) W(B) U(A) U(B)

- S(A) = shared lock on A
- X(A) = exclusive lock on A
- U(A) = unlock of A, or if more precision is needed:
  - US(A) = unlock shared lock of A
  - UX(A) = unlock exclusive lock of A

Issues with 2PL

- Recoverable schedules and Cascading rollbacks
- Deadlocks

Schedule with Aborted Transactions

- When a transaction aborts, the scheduler must undo its updates
- But some of its updates may have affected other transactions!

- T_1 R(V) W(V) Ab
- T_2 R(V) W(V) R(Y) W(Y) Co

- Cannot abort T_1 because cannot undo T_2

Recoverable Schedules

- A schedule is **recoverable** if:
  - It is conflict-serializable, and
  - Whenever a transaction T commits, all transactions that **have written elements** read by T have **already committed**

Recoverable Schedules

- A schedule is **recoverable** if:
  - It is conflict-serializable, and
  - Whenever a transaction T commits, all transactions that **have written elements** read by T have **already committed**

- T_1 R(V) W(V) Ab
- T_2 R(V) W(V) R(Y) W(Y) Co

Non-recoverable!

- T_1 R(V) W(V) Co
- T_2 R(V) W(V) R(Y) W(Y) Co

Recoverable!

Example

- T_1 R(X) W(X) Ab
- T_2 R(X) W(X) R(Y) W(Y)
- T_1 R(Y) W(Y) R(Z) W(Z)
- T_2 R(Z) W(Z) R(A) W(A)

Example

- T_1 R(X) W(X) Co
- T_2 R(X) W(X) R(Y) W(Y) Co
- T_1 R(Y) W(Y) R(Z) W(Z) Co
- T_2 R(Z) W(Z) R(A) W(A) Co

Recoverable schedule:
If T_j reads a value written by T_i then commit of T_j must be delayed after the commit of T_i

A schedule must always be recoverable!
But the above does not avoid cascading aborts

Cascading Aborts

- If a transaction T aborts, then we need to abort any other transaction T' that has read an element written by T.

- *Cascadeless schedule* **avoids cascading aborts** if whenever a transaction reads an element, the transaction that has **last written** it has **already committed**.
  - No dirty reads, thus abort or rollback does not cascade.
  - All cascadeless schedules are recoverable.

Example

- T_1 R(X) W(X) Co
- T_2 R(X) W(X) R(Y) W(Y) Co
- T_1 R(Y) W(Y) R(Z) W(Z) Co
- T_2 R(Z) W(Z) R(A) W(A) Co

Cascadeless schedule:
If T_j reads a value written by T_i then the read of of T_j is delayed after the commit of T_i

Does 2PL help with recoverable schedules?

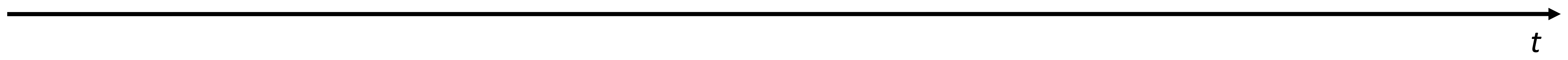
- T_1: $L_1(A)$ $L_1(B)$ R(A) W(A) $U_1(A)$
- T_2: $L_2(A)$R(A) W(A) $L_2(B)$ Denied

t

- T_1: R(B) W(B) $U_1(B)$ Ab
- T_2: Granted R(B) W(B) $U_2(A)$ $U_2(B)$ Co

Does 2PL help with recoverable schedules?

- T_1: $L_1(A)$ R(A) W(A)
- T_2: $L_2(A)$ Denied…



- T_1: $L_1(B)$ R(B) W(B) $U_1(A)$ $U_1(B)$ Rollback
- T_2: ….Granted R(A) W(A) $L_2(B)$

- T_1:
- T_2: R(B) W(B) $U_2(A)$ $U_2(B)$ Co

Cascades/Recoverable

- T_1 X(A) S(B) W(A)U(A) R(B) U(B) Co
- T_2 S(A) R(A)X(A)W(A)U(A)Co

- Is the schedule cascadeless?

- Is the schedule recoverable?

Cascades/Recoverable

- T_1 X(A) S(B) W(A)U(A) R(B) U(B) Co
- T_2 S(A) R(A)X(A)W(A)U(A) **Co**

- By delaying the commit of T2 we have made the schedule recoverable but not cascadeless.

Find which transactions need to be aborted?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T_1			R(B)											R(E)		W(B)	
T_2					W(A)					R(B)							
T_3	W(C)						R(B)						R(A)		R(D)		Ab
T_4		W(D)									W(A)						
T_5				R(C)				W(E)				W(C)					
T_6						W(E)			R(A)								

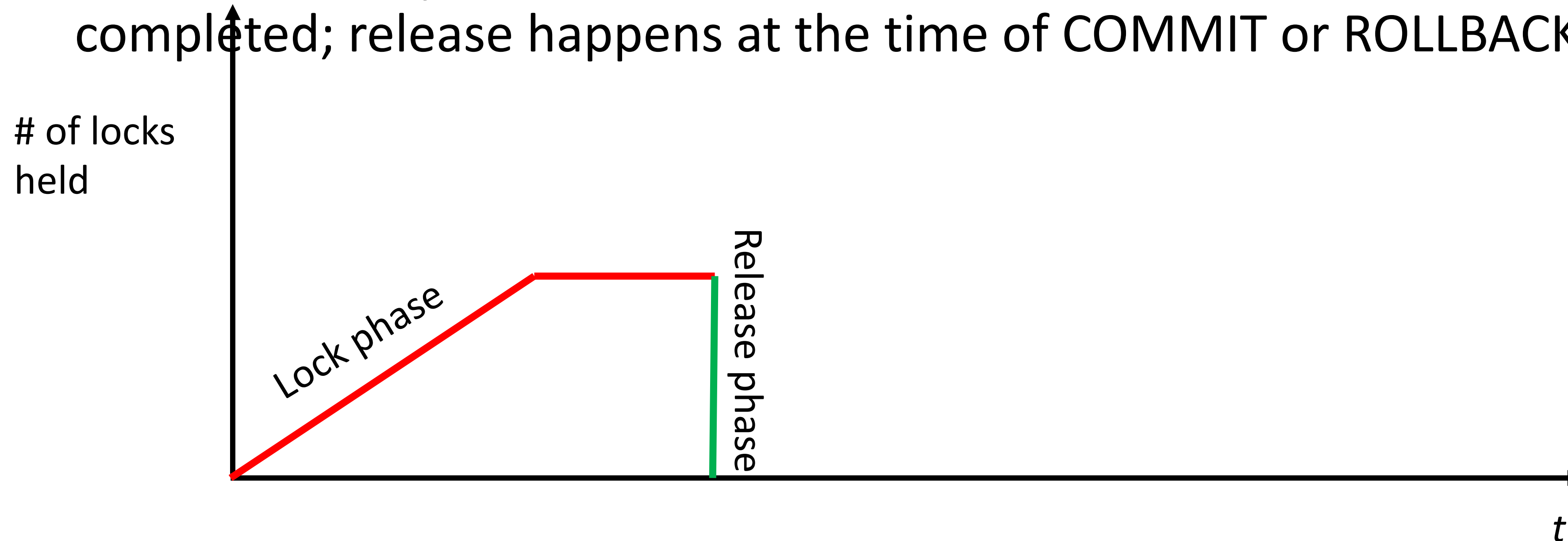
Find which transactions need to be aborted?

T_1			R(B)											R(E)		W(B)	
T_2					W(A)					R(B)							
T_3	W(C)						R(B)						R(A)		R(D)		Ab
T_4		W(D)									W(A)						
T_5				R(C)				W(E)				W(C)					
T_6						W(E)			R(A)								

T_3, T_5 T_1 must be rollbacked!

Strict 2PL

- Strict 2PL:
  - Like in 2PL, txans much acquire locks before reading or writing database objects.
  - All locks must be acquired before they are released.
  - All locks held by a transaction are released when the transaction is completed; release happens at the time of COMMIT or ROLLBACK



Summary of Strict 2PL

- Pros:
- Schedule is recoverable
- Schedule avoids cascading aborts
- Cons: deadlocks,
- General issues: implementation, lock modes, granularity, performance

Deadlocks

- Transaction 1
- xLock(A)
- Do something
- Lock(B)
- (waiting for T2 to unlock B)

- Transaction 2
- xLock(B)
- Do something
- Lock(A)
- (waiting for T1 to unlock A)

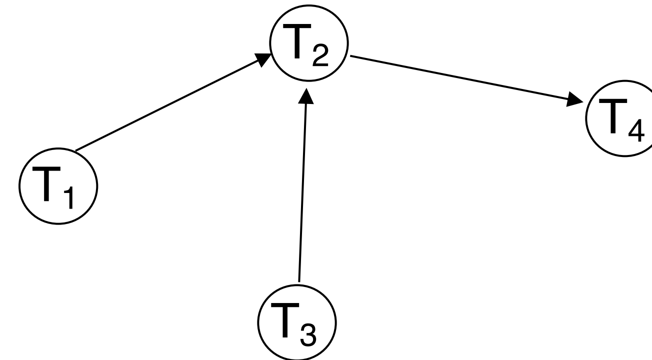
Deadlocks

T_1	T_2	T_3	T_4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	

Suppose T_4 requests lock-S(Z)....

Detecting Deadlocks using Wait-for-Graphs

- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from T_i to T_j
    - If T_i waiting for T_j

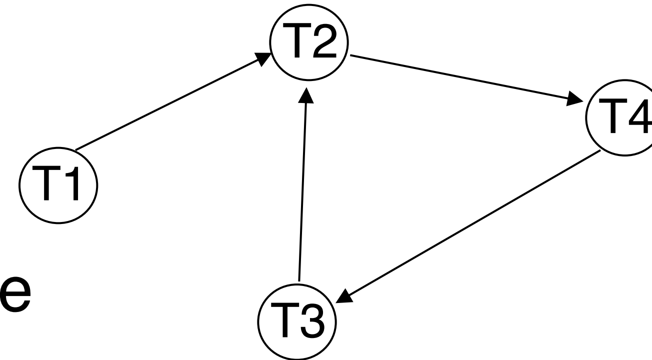


T_1	T_2	T_3	T_4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	

Detecting Deadlocks

- Wait-for graph has a cycle → deadlock

T_2, T_3, T_4 are deadlocked



•Build wait-for graph, check for cycle

•How often?

- Tunable

IF expect many deadlocks or many transactions involved THEN

run often to reduce aborts

ELSE run less often to reduce overhead

Recovering from Deadlocks

- Rollback transaction(s), but which ones?
  - Rollback the oldest txn, rollback the cheapest txn
  - Cons of oldest txn policy: Aborting old txns may cause a lot of computational investment to be thrown away.
    - Was it almost done?
    - How much will it have to redo?
    - Will it cause other rollbacks?
    - Is a partial rollback possible?
  - Cons of cheapest txn: Avoid starvation
    - Ensure same transaction not always chosen to break deadlock

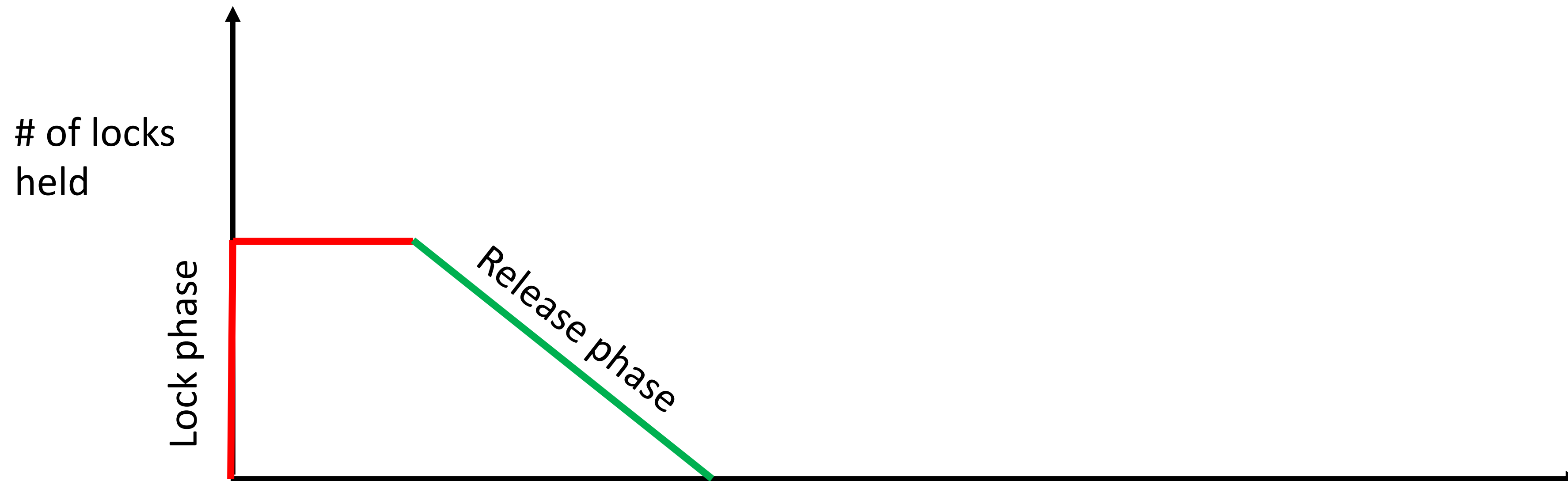
Deadlock Detection via Timeout

- Let transactions block on a lock request only for a limited time.
- After timeout, assume a deadlock has occurred and abort T.

- Policy is independent of transaction properties.

Preclaiming 2 Phase Locking Protocol

- All needed locks are declared at the beginning of the transaction.
- Advantage: No deadlocks! But rollbacks are cascading.
- Is it practical?



Example

T_1	R(V)			R(Z)		R(Y)	
T_2		R(Y)			W(V)		
T_3			W(V)				W(Z)

- Can it be achieved via 2PL?
- Can it be achieved via strict 2PL?

First Approximation

- T_1 S(V) R(V) S(Z)R(Z) S(Y)R(Y)
- T_2 S(Y)R(Y) X(V)W(V)
- T_3 X(V)W(V) X(Z) W(Z)

- Introduce shared locks for read statements and exclusive locks for write statements.

Example

• T_1	S(V) R(V)			S(Z)R(Z)		S(Y)R(Y)		
• T_2		S(Y)R(Y)			X(V)W(V)			
• T_3			X(V)W(V)				X(Z) W(Z)	

- Can schedule be achieved via 2PL? No. S(V) and X(V) conflict.
- Can schedule be achieved via strict 2PL? No. S(V) has to be held on till commit. Still conflicts with X(V)

First Approximation with unlocking?

- T_1 S(V) R(V) U(V) S(Z)R(Z) S(Y)R(Y)
- T_2 S(Y)R(Y) X(V)W(V)
- T_3 X(V)W(V) X(Z) W(Z)

First Approximation with unlocking?

- T_1 S(VZY) R(V) U(V) R(Z) R(Y)
- T_2 S(Y)R(Y) X(V)W(V)
- T_3 X(V)W(V) X(Z) W(Z)

First Approximation with unlocking?

- T_1 S(VZY) R(V) U(V) R(Z) U(Z) R(Y)
- T_2 S(Y)R(Y) X(V)W(V)
- T_3 X(V)W(V) X(Z) U(V) W(Z)

First Approximation with unlocking?

- T_1 S(VZY) R(V) U(V) R(Z) U(Z) R(Y) U(Y)
- T_2 S(Y)R(Y) X(V)W(V) U(VY)
- T_3 X(V)W(V) X(Z) U(V) W(Z) U(Z)

2PL serializable!

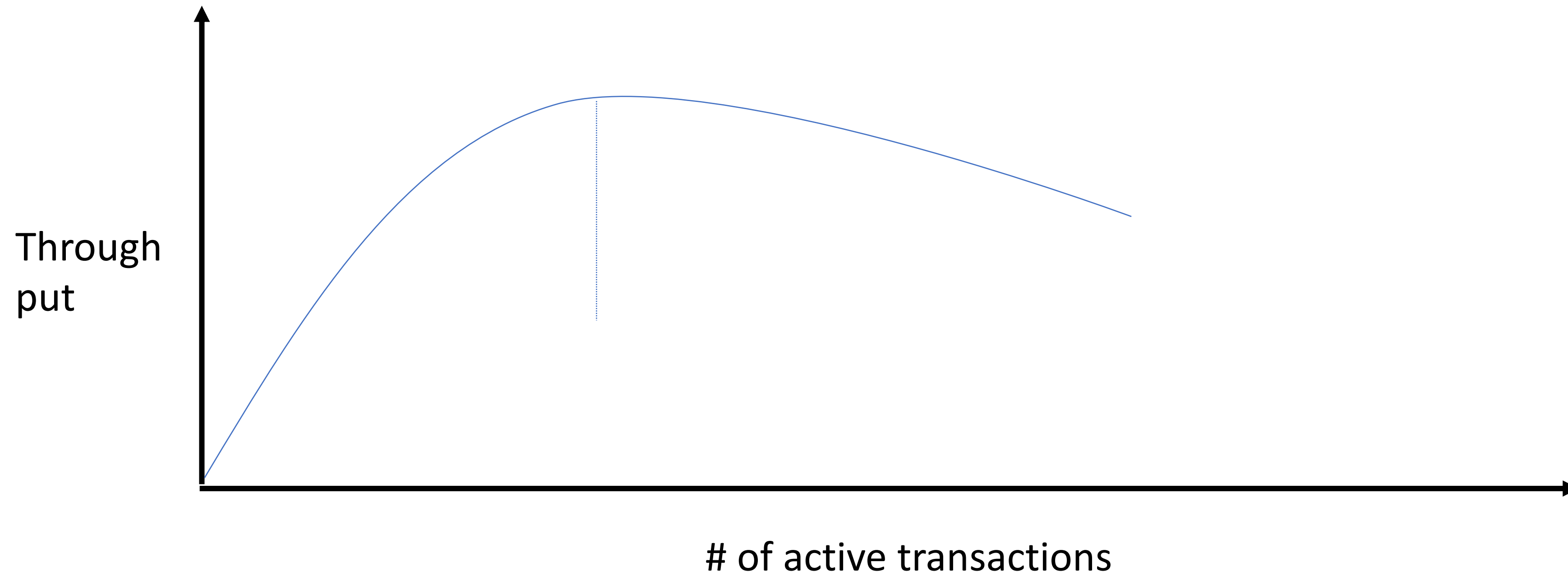
What about preclaiming 2PL?

- T_1 S(VZY) R(V) U(V) R(Z) U(Z) R(Y) U(Y)
- T_2 S(Y)R(Y) X(V)W(V) U(VY)
- T_3 X(V)W(V) X(Z) U(V) W(Z) U(Z)

2PL serializable!

Lock Performance

- Blocked txns or aborted txns
- Blocked txns lead to other txns waiting
- Aborted txns waste work already done.



What do real DBs do?

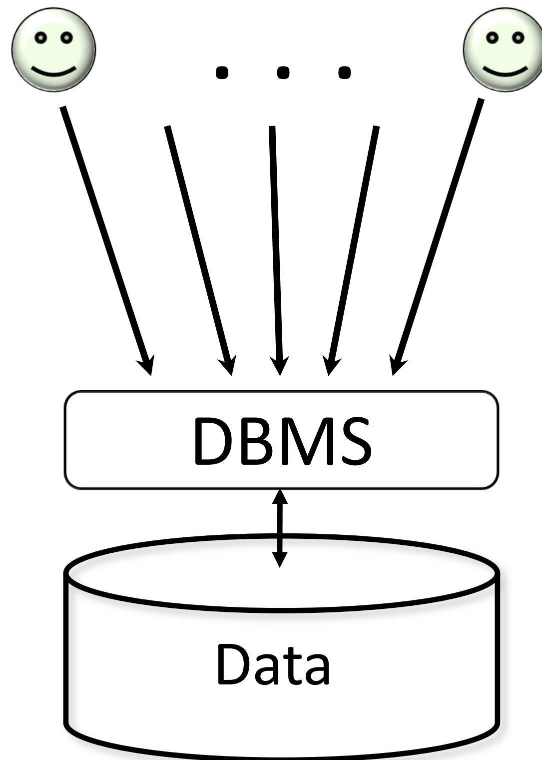
- Perfect isolation so far but do real DBs allow weaker consistency

Isolation Levels

- Some degree of inconsistency may be acceptable for specific applications to gain increased concurrency and performance

- E.g. accept inconsistent read anomaly and be rewarded with improved concurrency. Relaxed inconsistency guarantees can lead to increased throughput.

SQL Directives for managing transactions



Weaker "Isolation Levels"

`Read Uncommitted`

`Read Committed`

`Repeatable Read`

Strongest "Isolation Levels"

`Serializable order`

↓ Overhead ↑ Concurrency

↓ Consistency Guarantees

Isolation Levels in SQL

- “Dirty reads”

SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED

- “Committed reads”

SET TRANSACTION ISOLATION LEVEL READ COMMITTED

- “Repeatable reads”

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ

- Serializable transactions

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

Read uncommitted

- A transaction may perform dirty reads

```
Select GPA from Student Where sizeHS > 2500
```

```
Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500
```

concurrent with …

```
Set Transaction Isolation Level Read Uncommitted;
Select Avg(GPA) From Student;
```

Read committed

- A transaction will *not* perform dirty reads
- Only reads commited values of other transactions

Still does not guarantee global serializability

```
Select GPA From Student Where sizeHS > 2500
```

```
Update Student Set GPA = (1.1) * GPA Where sizeHS > 2500
```

concurrent with …

```
Set Transaction Isolation Level Read Committed;
Select Avg(GPA) From Student;
Select Max(GPA) From Student;
```

Repeatable read

- A transaction will not perform dirty reads
- An attrubute read multiple times cannot change value

Still does not guarantee global serializability

```
Update Student Set GPA = (1.1) * GPA;
Update Student Set sizeHS = 1500 Where sID = 123;
```

concurrent with …

```
Set Transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;
Select Avg(GPA) From Student;
Select Avg(sizeHS) From Student;
```

Repeatable read-non serializable

- An item read multiple times cannot change value
  But a relation *can* change: due to unlocked tuples that are inserted/or locked tuples that are deleted

```
Insert Into Student [ 100 new tuples ]
```

concurrent with …

```
Set Transaction Isolation Level Repeatable Read;
Select Avg(GPA) From Student;
Select Max(GPA) From Student;
```

With locking

- T_1 locks all rows
- T_2 locks the new rows
- T_2 releases its lock
- T_2 reads new row too!

- Both transactions properly follow 2PL protocol!
- Nevertheless T_1 observed an effect caused by T_2.
  - Isolation violated!
  - Cause of the problem: T_1 can only lock existing rows!

Phantom Problem

- A “phantom” is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

- In our example:
  - T1: reads
  - T2: inserts
  - T1: re-reads: a new tuple appears !

Phantom Problem

- In a static database:
  - Conflict serializability implies serializability
- In a dynamic database, this may fail due to phantoms
- Strict 2PL guarantees conflict serializability, but not serializability

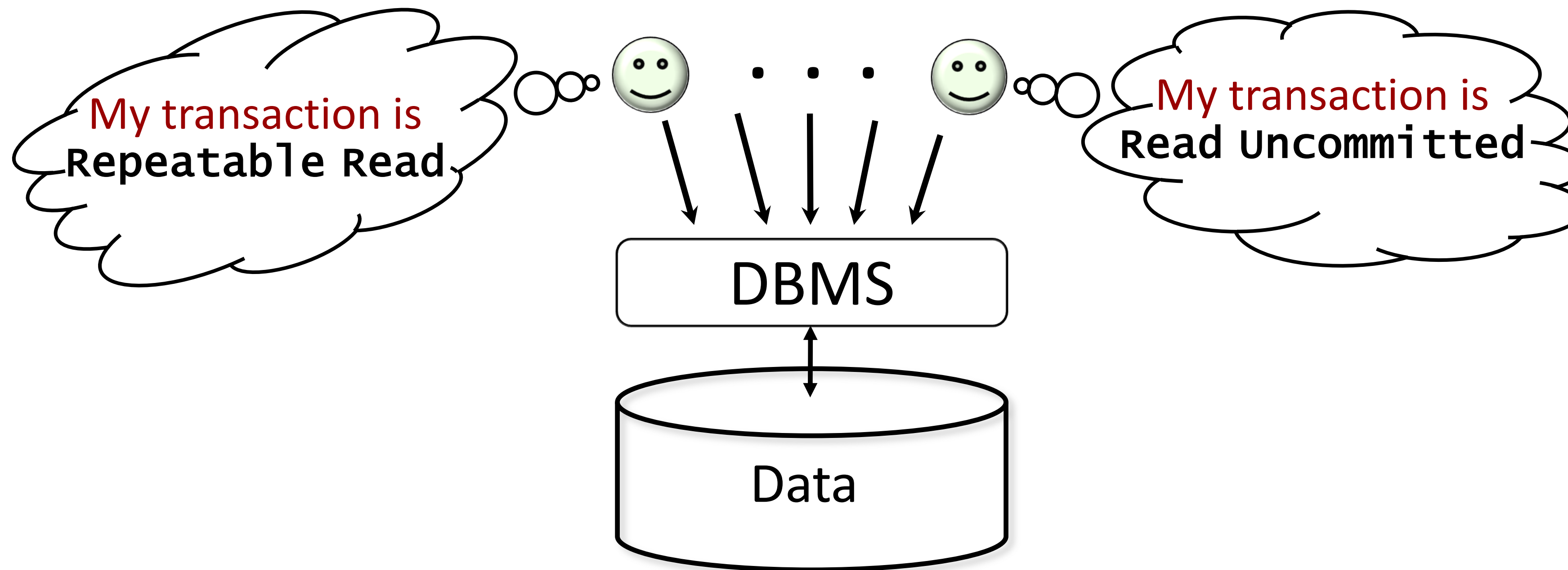
Dealing with Phantoms

- Lock the entire table, or
- Lock the index entry for ‘blue’
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

- Dealing with Phantoms is Expensive!

Isolation LEVELS

- Per transaction
- “In the eye of the beholder”
- Affect applies to read statements



Isolation Level: Dirty Reads Variation of 2PL

- “Long duration” WRITE locks
  - Strict 2PL
- No READ locks
  - Read-only transactions are never delayed

- Possible problems:
  - dirty and inconsistent reads

Isolation Level: Read Committed Variation of 2PL

- "Long duration" WRITE locks
  - Strict 2PL
- "Short duration" READ locks
  - Only acquire lock while reading (not 2PL)

- Unrepeatable reads
  - When reading same element twice, may get two different values

Isolation Level: Repeatable Read

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL

- This is not serializable yet !!!
- Why? Phantom

Isolation Level Serializable

- “Long duration” WRITE locks
  - Strict 2PL
- “Long duration” READ locks
  - Strict 2PL
- Predicate locking
  - To deal with phantoms

Transactions-summary

- **Serializable**
  - Strongest isolation level
  - SQL Default
- **Read Uncommitted**
  - A data item is dirty if it is written by an uncommitted transaction.
  - Problem of reading dirty data written by another uncommitted transaction: what if that transaction eventually aborts?
- **Read Committed**
  - Cannot read dirty data written by other uncommitted transactions.
  - But read-committed is still not necessarily serializable
- **Repeatable Read**
  - If a tuple is read once, then the same tuple must be retrieved again if query is repeated.
  - Still not serilizable; may see phantom tuples—tuples inserted by other concurrent transactions

SQL-92 Isolation Level and Consistency Gurantees

Isolation level	Dirty read	Non-repeatable read	Phantom rows
Read uncommitted	possible	possible	possible
Read committed	Not possible	possible	possible
Repeatable read	Not possible	Not possible	possible
Serializable	Not possible	Not possible	Not possible

Different databases support different isolation level.

READ-ONLY Transactions

- Client 1: START TRANSACTION

INSERT INTO SmallProduct(name, price)
SELECT pname, price FROM Product WHERE price <= 0.99
DELETE FROM Product WHERE price <=0.99
COMMIT

- Client 2: SET TRANSACTION READ ONLY

START TRANSACTION
SELECT count(*) FROM Product
SELECT count(*) FROM SmallProduct
COMMIT

- Always check documentation!
- DB2: Strict 2PL
- SQL Server:
  - Strict 2PL for standard 4 levels of isolation
  - Multiversion concurrency control for snapshot isolation
- PostgreSQL: Snapshot isolation; recently: seralizable Snapshot isolation (!)
- Oracle: Snapshot isolation

Lab 3

- NO STEAL / FORCE buffer management policy
  - you shouldn't evict dirty(updated) pages from the buffer pool if they are locked by an uncommitted transaction. (this is NO STEAL)
  - on transaction commit, you should force dirty pages to disk. (e.g., write the pages out) (this is FORCE)
- Locking at page level
  - you acquire and release locks in BufferPool.getPage(), instead of adding calls to each of your operators
  - Might have to change previous implementations to access pages using BufferPool.getPage()

Lab 3

- You need to implement shared and exclusive locks
  - Before read, it must have a shared lock
  - Before write, it must have an exclusive lock
  - Multiple transactions can have a shared lock
  - Only one transaction may have an exclusive lock on an object
  - If transaction t is the only transaction holding a shared lock on an object o, t may upgrade its lock on o to an exclusive lock
  - Logic provided in comments
- You need to implement strict two-phase locking
  - transactions should acquire the appropriate type of lock on any object before accessing that object
  - transaction shouldn’t release any locks until after the transaction commits.

Lab 3

- A LockManager bare bones class is provided which you will need to interface with BufferManager
  - The class will hold data structures to keep track of which locks each transaction holds and that check to see if a lock should be granted to a transaction when it is requested.
- Read about Synchronization in Java, and check the use of synchronized keyword in appropriate places in LockManager
  - You will have to also throw appropriate exceptions like TransactionAbortedException

Lab 3

- Handling deadlocks
  - implement a simple timeout policy that aborts a transaction if it has not completed after a given period of time
  - implement a cycle-detection in a dependency graph data structure, if cycle exists when granting a new lock abort something.
- Design Choices:
  - Locking Granularity: page-level (the tests also assume page-level)
  - Deadlock Detection: timeout vs dependency graphs
  - Deadlock Resolution: aborting yourself vs aborting others
  - Read the spec carefully for more details about various methods and edge cases