Discussion 8

Transactions & Concurrency

Announcements

Vitamin 8 (Transactions & Concurrency) due **Monday, March 18 at 11:59pm**

Project 3 Part 2 (Joins & QO) due **Wednesday, March 13 at 11:59pm**

Mid-Semester Survey due Monday, March 11 at 11:59pm

Agenda

I. Transactions
II. Serializability
III. Locking
IV. Multi-granularity Locking
V. Worksheet

Transactions

Transactions

- Collections of operations that are a single logical unit
  - e.g. *swapping sections on CalCentral* is a single logical operation, but consists of:
    - checking if the new section has space
    - removing student from old section
    - adding student to new section
  - We would like these components to all happen at once
    - don’t want another student to fill up new section after removing student from old section!

Transactions

- We want transactions to obey **ACID**
  - **atomicity**: *all* operations in a transaction happen, or *none* of them
  - **consistency**: database consistency (unique constraints, etc) is maintained
  - **isolation**: should look like we only run 1 transaction at a time (even if we run multiple concurrently)
  - **durability**: once a transaction commits, it persists
- **commit**: indicates *successful* transaction (save changes)
- **abort**: indicates *unsuccessful* transaction (revert changes)

Transactions

- In the context of swapping sections on Calcentral....
  - **atomicity**: either drop *and* add, or no change - shouldn't drop student from old section then not add into new section!
  - **consistency**: make sure constraints like "student only enrolled in one section" are maintained
  - **isolation**: should look like we're not making other changes at the same time - for example, enrolling Amy into section after dropping Carl but before adding Carl to new section
  - **durability**: once we say that we've made the change, student should not suddenly find themselves in old section later on!

Transactions

- For this week, we'll focus on **isolation**. Durability and atomicity will be covered in the future
  - **isolation**: should look like we only run 1 transaction at a time (even if we run multiple concurrently)

Concurrency

- There are benefits to running many transactions *concurrently* (at the same time)
    - **throughput argument**: can increase processor/disk utilization (e.g. work on another transaction while one is busy waiting for data from disk) ➔ *more transactions done per second*
    - **latency argument**: if we run lots of transactions at once, a long running transaction doesn't cause short transactions to wait a long time to start ➔ *transactions may get finished faster*
- but we still want to maintain isolation!

Serializability

Serializability

- **schedule**: order in which we execute operations of a set of transactions
- **serial schedule**: every transaction runs start to finish without any **interleaving**
- We say two schedules are **equivalent** if:
    - They involve the same transactions
    - Each transaction has its operations in the same order
    - The final state after all the transactions is the same
- Having isolation basically means having a schedule that is equivalent to a serial schedule (**serializable**)

Serializability

- How do you check if a schedule is serializable?
  - You don’t - even checking if a schedule is view serializable (stronger condition than serializable - some serializable schedules are not view serializable) is NP-complete
- We instead check if schedules are **conflict serializable**
  - Much stronger condition than serializable - lots of serializable schedules are not conflict serializable
  - but usually good enough

Conflict Serializability

- Two operations in a schedule **conflict** if:
  - at least one operation is a write
  - they are on *different* transactions
  - they work on the *same* resource
- Conflicts are essentially pairs of operations that we need to be careful about

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

Conflict Serializability

- Two operations in a schedule **conflict** if:
  - at least one operation is a write
  - they are on *different* transactions
  - they work on the *same* resource
- Conflicts are essentially pairs of operations that we need to be careful about

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

stale read: T_1 uses old value of B - if reversed order, output may differ

Conflict Serializability

- Two schedules are **conflict equivalent** (and therefore equivalent) if every conflict is ordered the same way
  - *two reads* are not a conflict, so they can be ordered in any way
  - two operations in the *same transaction* do not conflict, because the order is fixed anyways
  - two operations on *different resources* are not a conflict, and can happen in any order
- A schedule is **conflict serializable** if it is conflict equivalent to a serial schedule

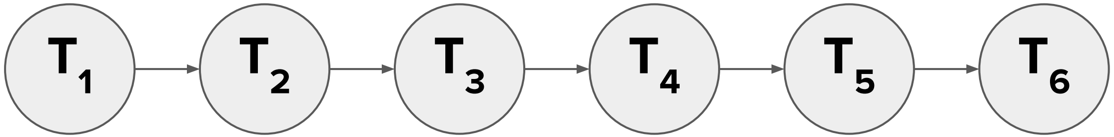
Conflict Serializability

- How do we check for conflict equivalence/serializability?
  - We build a **dependency graph (precedence graph)**
    - One node per transaction
    - If an operation in T_i conflicts with an operation in T_j, and the operation in T_i comes first, add an edge from T_i to T_j

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

Conflict Serializability

- To find equivalent schedules, do a topological sort of the graphs!
  - If all the arrows are in the same direction, the two schedules are conflict equivalent
    - A serial schedule's graph looks like:



    - No cycles in a serial schedule's dependency graph
    - Theorem: all conflict serializable schedules have an acyclic dependency graph
      - We can just check if graph has a cycle!

View Serializability

- Conflict serializable cares about the order of **blind writes**
  - intermediate writes that are overwritten without an interleaving read
  - order of blind writes doesn’t actually matter for serializability
- Two schedules are **view equivalent** if they are conflict equivalent, except for potentially reordering blind writes
- A schedule is **view serializable** if it is view equivalent to a serial schedule
- NP-complete to check…
  - Also doesn’t cover all serializable schedules!

View Serializability

- How do we check for view equivalence/serializability?
  - Same initial reads
    - Transaction reads in the same initial value for X in both schedules
  - Same dependent reads
    - If transaction reads in a value X written by another transaction in one schedule, it also reads value of X written by same transaction in view equivalent schedule.
  - Same winning writes
    - Same transaction writes the final value of X in both schedules

Schedule 1

T_1	R(A)		W(A)	
T_2		W(A)		
T_3				W(A)

Schedule 2

T_1	R(A)	W(A)		
T_2			W(A)	
T_3				W(A)

View Serializability

- **Blind Writes**: writes with no reads in between
  - Shown in the sequence of three writes in both schedules
- How do we check for view equivalence/serializability?
  - Same initial reads
  - Same dependent reads
  - Same winning writes

Schedule 1

T_1	R(A)		W(A)	
T_2		W(A)		
T_3				W(A)

Schedule 2

T_1	R(A)	W(A)		
T_2			W(A)	
T_3				W(A)

View Serializability

- How do we check for view equivalence/serializability?
  - **Same initial reads**
    - Transaction reads in the same initial value for X in both schedules
  - Same dependent reads
    - If transaction reads in a value X written by another transaction in one schedule, it also reads value of X written by same transaction in view equivalent schedule.
  - Same winning writes
    - Same transaction writes the final value of X in both schedules

Schedule 1

T_1	**R(A)**		W(A)	
T_2		W(A)		
T_3				W(A)

Schedule 2

T_1	**R(A)**	W(A)		
T_2			W(A)	
T_3				W(A)

View Serializability

- How do we check for view equivalence/serializability?
  - Same initial reads
    - Transaction reads in the same initial value for X in both schedules
  - Same dependent reads
    - If transaction reads in a value X written by another transaction in one schedule, it also reads value of X written by same transaction in view equivalent schedule.
  - **Same winning writes**
    - Same transaction writes the final value of X in both schedules

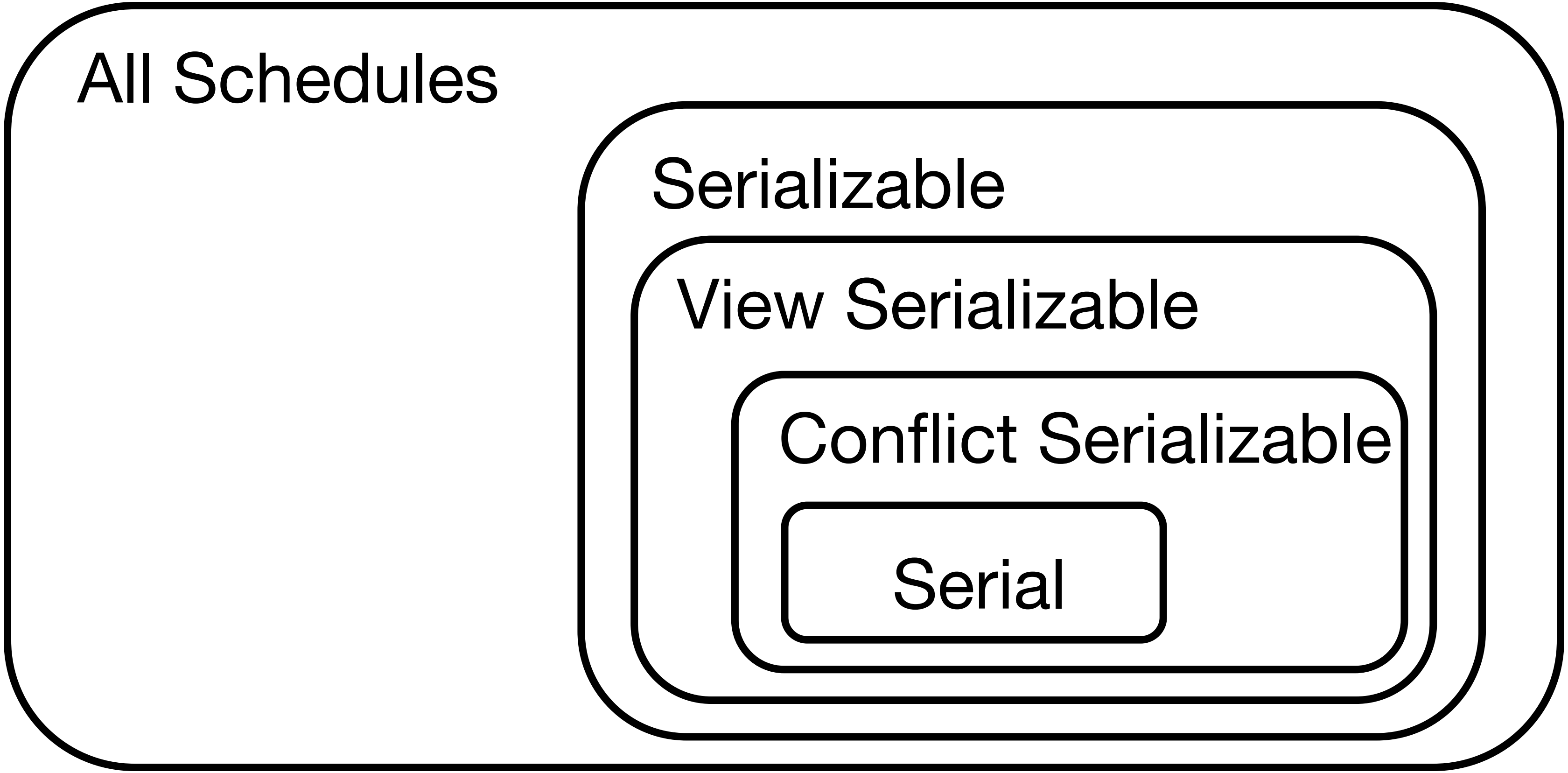
Schedule 1

T_1	R(A)		W(A)	
T_2		W(A)		
T_3				**W(A)**

Schedule 2

T_1	R(A)	W(A)		
T_2			W(A)	
T_3				**W(A)**

Types of Serializability



Phantom Problem

- Phantom problem: occurs within a single transaction when the same query produces different sets of rows at different times
  - example: T1 executes SELECT twice, but returns a row the second time that was not returned the first time because T2 inserts a new value between the two SELECT statements → “phantom” row
  - these types of schedules are not serializable

Serializability Summary

- **Serial:** schedule occurs in isolation (as if it’s the only one running)
- **Serializable:** schedule appears to occur as if in isolation (looks equivalent to a serial schedule)
- **Conflict Serializable:** All conflicting operations are ordered in the same way as that of a serial schedule
  - What we focus on in this course!
  - Conflict serializability implies serializability
- **View Serializable:** All conflicts are conflict equivalent (just like conflict serializable) to a serial schedule but blind writes may appear in any order

Question 1
Conflict Serializability

Conflict Serializability

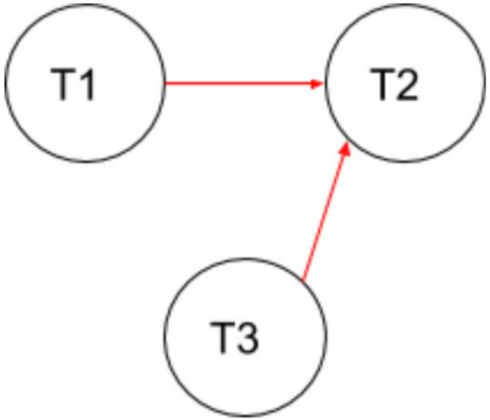
T_1		R(A)	W(A)	R(B)					
T_2					W(B)	R(C)	W(C)	W(A)	
T_3	R(C)								W(D)

(a) Draw the dependency graph (precedence graph) for the schedule.

Conflict Serializability

T_1		R(A)	W(A)	R(B)					
T_2					W(B)	R(C)	W(C)	W(A)	
T_3	R(C)								W(D)

(a) Draw the dependency graph (precedence graph) for the schedule.



Conflict Serializability

T_1		R(A)	W(A)	R(B)					
T_2					W(B)	R(C)	W(C)	W(A)	
T_3	R(C)								W(D)

(b) Is this schedule conflict serializable? If so, what are all the conflict equivalent serial schedules? If not, why not?

Conflict Serializability

T_1		R(A)	W(A)	R(B)					
T_2					W(B)	R(C)	W(C)	W(A)	
T_3	R(C)								W(D)

(b) Is this schedule conflict serializable? If so, what are all the conflict equivalent serial schedules? If not, why not?

Conflict equivalent to T_3, T_1, T_2 and T_1, T_3, T_2
(possible topological sorts of the dependency graph).

Conflict Serializability

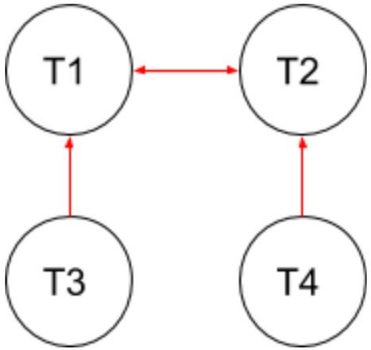
T_1	R(A)		R(B)				W(A)	
T_2		R(A)		R(B)				W(B)
T_3					R(A)			
T_4						R(B)		

(c) Draw the dependency graph (precedence graph) for the schedule.

Conflict Serializability

T_1	R(A)		R(B)				W(A)	
T_2		R(A)		R(B)				W(B)
T_3					R(A)			
T_4						R(B)		

(c) Draw the dependency graph (precedence graph) for the schedule.



Conflict Serializability

T_1	R(A)		R(B)				W(A)	
T_2		R(A)		R(B)				W(B)
T_3					R(A)			
T_4						R(B)		

(d) Is this schedule conflict serializable? If so, what are all the conflict equivalent serial schedules? If not, why not?

Conflict Serializability

T_1	R(A)		R(B)				W(A)	
T_2		R(A)		R(B)				W(B)
T_3					R(A)			
T_4						R(B)		

(d) Is this schedule conflict serializable? If so, what are all the conflict equivalent serial schedules? If not, why not?

No, there's a cycle between T_1 and T_2 in the dependency graph.

Locking

Simple Locking

- Database operations tend to include more reads than writes, so we use readers-writers locks
- A transaction may lock a resource in one of two ways: **S**hared or e**X**clusive, corresponding to reading and writing
  - An **S** lock lets a transaction **read** a resource (e.g. tuple)
    - Many transactions can hold S locks on a resource at once
  - An **X** lock lets a transaction **modify** a resource
    - No other transaction can have *any* type of lock while a transaction has an X lock
- If a transaction can't get a lock it wants, it blocks, and waits until another transaction releases the conflicting lock

Simple Locking

- The **lock compatibility matrix** tells us whether multiple transactions can acquire locks on the same resource at the same time
  - Consider (S,X) = false. This means that it is not possible for T_1 to hold an S lock on a resource while T_2 holds an X lock on the same resource.

	NL (no lock held)	S	X
NL (no lock held)	✔	✔	✔
S	✔	✔	❌
X	✔	❌	❌

Deadlocks

- What if T_1 is waiting for T_2 to release a lock, but T_2 is also waiting for T_1 to release a lock?
    - This is called a **deadlock** - when a bunch of transactions are waiting on each other in a cycle
- If T1 is waiting for T2 to release a lock, T1 cannot do any more operations or acquire more locks while it is waiting
- We can either avoid them in the first place (**deadlock avoidance**) or catch them (**deadlock detection**) and abort a transaction in the deadlock

Deadlock Avoidance

- Two approaches
  - **wait-die**: if transaction T_i wants a lock but T_j holds a conflicting lock
    - if T_i is higher priority, it *waits* for T_j to release conflicting lock
    - if T_i is lower priority, it aborts ("*die*")
    - transactions can only wait on lower priority transactions ➔ cannot have deadlock (lowest priority transactions cannot wait)
  - **wound-wait**: if transaction T_i wants a lock but T_j holds a conflicting lock
    - if T_i is higher priority, it causes T_j to abort ("*wound*")
    - if T_i is lower priority, it *waits* for T_j to finish
    - transactions can only wait on higher priority transactions ➔ cannot have deadlock (highest priority transactions cannot wait)

Deadlock Avoidance

- Unless priority is explicitly defined, assign a **transaction's priority by age: now - start time**
- While T_1 *waits* for T_2 to release a lock, T_1 cannot do any more operations or acquire other locks

Deadlock Detection

- We draw out a **"waits-for" graph**
  - One node for each transaction
  - If T_i *holds* a lock that conflicts with the lock that T_j wants (or T_j "waits for" T_i), we add an edge from T_j to T_i

  - A cycle indicates a deadlock (between the transactions in the cycle) - we can abort one to end the deadlock
- Alternative approach (used in some real databases): just kill transactions if they aren't doing anything for a while

Question 2
Deadlocks

Deadlocks

T_1	S(A)	S(D)		S(B)					
T_2			X(B)				X(C)		
T_3					S(D)	S(C)			X(A)
T_4								X(B)	

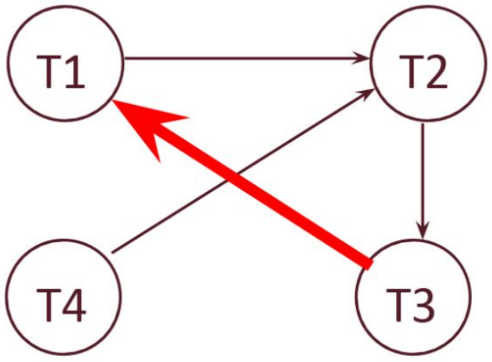
(a) Draw a “waits-for” graph and state whether or not there is a deadlock.

Deadlocks

T_1	S(A)	S(D)		S(B)					
T_2			X(B)				X(C)		
T_3					S(D)	S(C)			X(A)
T_4								X(B)	

(a) Draw a “waits-for” graph and state whether or not there is a deadlock.

Deadlock between T_1, T_2, T_3.



Deadlocks

T_1	S(A)	S(D)		S(B)						
T_2			X(B)				X(C)			
T_3					S(D)	S(C)			X(A)
T_4								X(B)		

(b) If we try to avoid deadlock by using wait-die deadlock avoidance policy, would any transactions be aborted? Assume T1 priority > T2 > T3 > T4.

Deadlocks

T_1	S(A)	S(D)		S(B)					
T_2			X(B)				X(C)		
T_3					S(D)	S(C)			X(A)
T_4								X(B)	

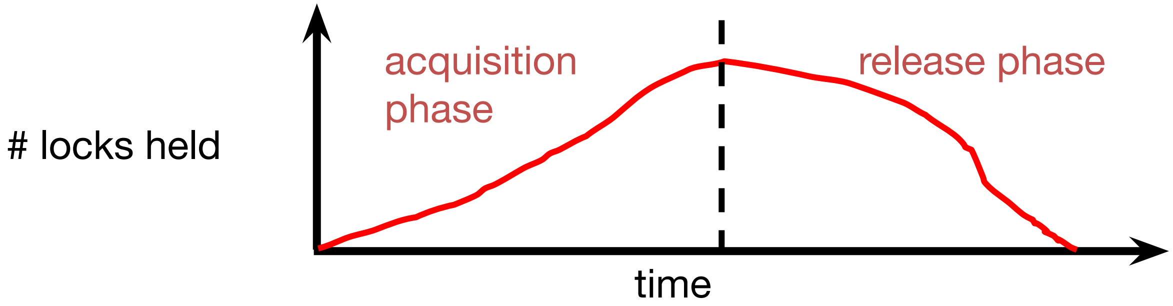
(b) If we try to avoid deadlock by using wait-die deadlock avoidance policy, would any transactions be aborted? Assume T1 priority > T2 > T3 > T4.

Yes, T3 and T4 are aborted. When T4 attempts to acquire a lock on B, which is held by T2, T4 will abort since it is attempting to acquire a lock held by a transaction with higher priority. The same thing happens when T3 attempts to acquire a lock on A, which is held by T1.

Locking (2PL and Strict 2PL)

2-Phase Locking (2PL)

- One way to enforce conflict serializability
- In **2-phase locking**,
  - a transaction may not acquire a lock after it has released any lock
  - two “phases”
    - from start to until a lock is released, the transaction is only acquiring locks
    - then until the end of the transaction, it is only releasing locks



Cascading Aborts

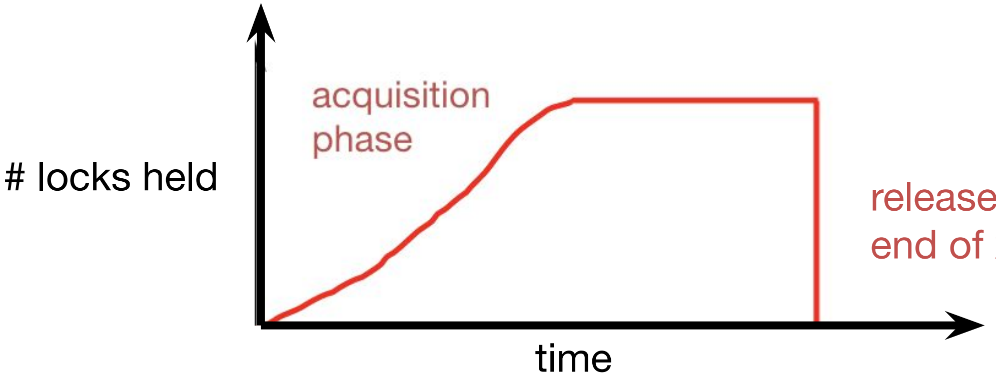
- Consider the following (+ for acquire, - for release):

T_1	+X(A)	-X(A)						abort!
T_2			+S(A)					
T_3			+S(A)	+X(B)	-X(B)			
T_4						+S(B)		
T_5							+S(B)	

- T_1 aborting means *all* of the transactions have to be rolled back
  - Even though T_4 and T_5 didn't read A at all
  - This is a **cascading abort**

Strict 2-Phase Locking (Strict 2PL)

- The problem is that 2PL lets another transaction read new values before the transaction commits (since locks can be released long before commit)
- **Strict 2PL** avoids cascading aborts
  - Same as 2PL, except only allow releasing locks at end of transaction



Strict 2PL and Cascading Aborts

T_1	+X(A)	-X(A)						abort!
T_2			+S(A)					
T_3			+S(A)	+X(B)	-X(B)			
T_4						+S(B)		
T_5							+S(B)	

- The previous schedule would have been illegal under Strict 2PL
  - T_1 would not release its lock on A until it commits
  - Other transactions would not have seen A and would not have to be rolled back

Question 3
Locking

Locking

(a) What is printed, assuming we initially have B = 3 and F = 300?

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(a) What is printed, assuming we initially have B = 3 and F = 300?

3030

T1: Lock X(B) → Read B → B = 3*10 = 30 → Write B → Lock X(F) → Unlock B

T2: Lock S(F) (waiting for T1 to release X Lock on F)

T1: F = 30*100 = 3000 → Write F → Commit → Unlock F (S Lock on F is granted to T2)

T2: Read F → Unlock F → Lock S(B) → Read B → Compute and print F + B = 3000 + 30 = 3030 → Commit → Unlock B

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(b) Does the execution use 2PL, strict 2PL, or neither?

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(b) Does the execution use 2PL, strict 2PL, or neither?

Neither - S(F) unlocked before T_2 acquires S(B)

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(c) Would moving Unlock F in T_2 to any point after Lock S(B) change this (or keep it) in 2PL?

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(c) Would moving Unlock F in T_2 to any point after Lock S(B) change this (or keep it) in 2PL?

Yes - all locks would be acquired (for T_2) before any are released.

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(d) Would moving Unlock F in T_1 and Unlock F in T_2 to the end of their respective transactions change this (or keep it) in strict 2PL?

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(d) Would moving Unlock F in T_1 and Unlock F in T_2 to the end of their respective transactions change this (or keep it) in strict 2PL?

No - T_1 still unlocks B before the end of the transaction

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(e) Would moving Unlock B in T_1 and Unlock F in T_2 to the end of their respective transactions change this (or keep it) in strict 2PL?

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Locking

(e) Would moving Unlock B in T_1 and Unlock F in T_2 to the end of their respective transactions change this (or keep it) in strict 2PL?

Yes - all unlocks would only happen when the respective transactions end

T_1	T_2
Lock X(B)	
Read B	
B := B * 10	
Write B	
Lock X(F)	
Unlock B	
	Lock S(F)
F := B * 100	
Write F	
Commit	
Unlock F	
	Read F
	Unlock F
	Lock S(B)
	Read B
	Print F+B
	Commit
	Unlock B

Multi-granularity Locking

Multi-granularity Locking

- What are the granularity of our locks?
  - **Tuple-level**: one lock per tuple, but then a scan might lock millions of locks → *high locking overhead*
  - **Table-level**: one lock per table, but then any update locks the entire table → *low concurrency*
- **Multi-granularity locking**: instead of picking one, allow for locking at different levels of granularity (e.g. database, table, page, tuple)
  - scans can acquire a lock for entire table → low overhead
  - updating tuple can acquire a lock for tuple only → high concurrency

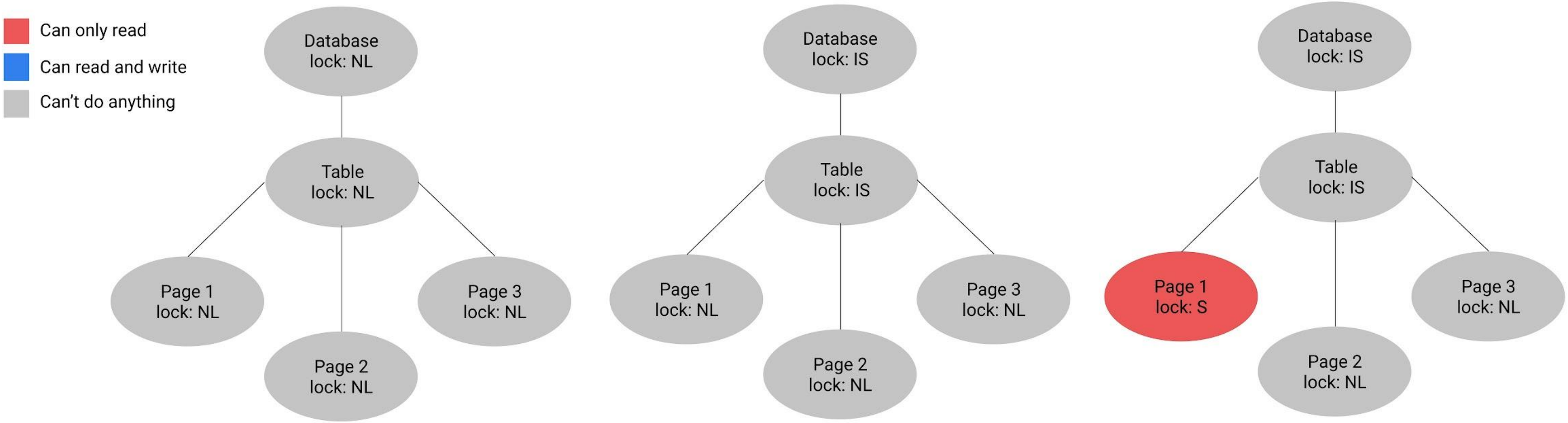
Multi-granularity Locking

- If a transaction has X on a tuple, it shouldn't be possible to get S on the entire table
  - checking locks on all tuples → high locking overhead again
- For each lock, require that transactions hold appropriate **intent locks** at all *higher levels* of granularity
  - need an intent lock on database, table, page to hold S/X lock on tuple
  - intent lock tells us about any potential locking at a lower level

Multi-granularity Locking

- Two new lock types:
  - IS = **I**ntent to acquire **S**hared lock at a lower level
  - IX = **I**ntent to acquire e**X**clusive lock at a lower level
- Require that a transaction has IS to acquire S, IX to acquire X
  - To get S(tuple), need: IS(database), IS(table), IS(page)
    - IX lets you acquire S/IS locks too
  - If a transaction tries to get S(table), and sees another transaction has IX(table) → another transaction has X lock on a tuple, so we can't get S(table) yet

Multi-granularity Locking



Request S lock on Page 1

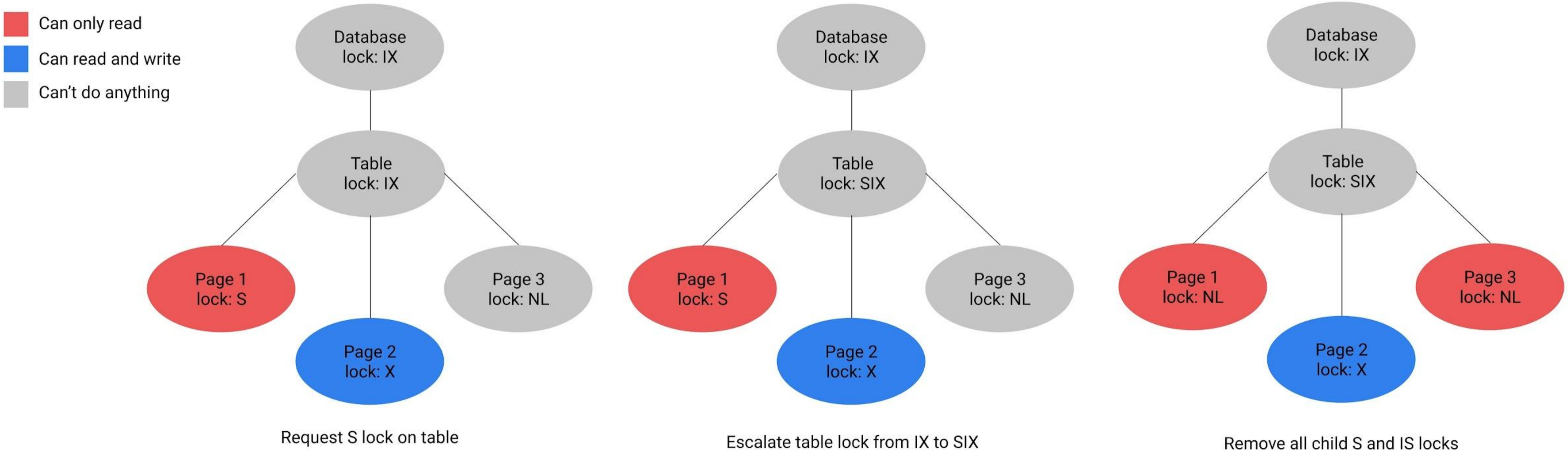
Request IS lock on all ancestors, substituting as necessary

Acquire S lock on Page 1

Multi-granularity Locking

- What if we want to scan table AND update a page?
  - X on entire table → low concurrency
  - IX on table, S/X locks on every page → high locking overhead
- One more lock type:
  - SIX = **S**hared + **I**ntent to acquire e**X**clusive lock at a lower level
    - Basically equivalent to having both S and IX locks – can read entire table, can also acquire X locks

Multi-granularity Locking



Multi-granularity Locking

- Our new compatibility matrix

	NL	IS	IX	SIX	S	X
NL	✓	✓	✓	✓	✓	✓
IS	✓	✓	✓	✓	✓	✗
IX	✓	✓	✓	✗	✗	✗
SIX	✓	✓	✗	✗	✗	✗
S	✓	✓	✗	✗	✓	✗
X	✓	✗	✗	✗	✗	✗

Multi-granularity Locking

- Parent ➜ Children Lock Relationships for a Single Transaction

Parent Lock	**Possible Children Locks**
NL	Nothing
S	Nothing
X	Nothing
IS	IS, S
IX	IX, X, IS, S, SIX
SIX	IX, X

Question 4
Multi-granularity Locking

Multi-granularity Locking

(a) Suppose a transaction T_1 wants to scan a table R and update a few of its tuples. What kinds of locks should T_1 have on R, the pages of R, and the updated tuples?

Multi-granularity Locking

(a) Suppose a transaction T_1 wants to scan a table R and update a few of its tuples. What kinds of locks should T_1 have on R, the pages of R, and the updated tuples?

SIX on R

IX on page (no need for SIX since already have S on R)

X on tuples being modified

Multi-granularity Locking

(b) Is an S lock compatible with an IX lock?

Multi-granularity Locking

(b) Is an S lock compatible with an IX lock?

Suppose T_1 wants S(A) and T_2 wants IX(A).

S(A) implies T_1 will read all of A.

IX(A) implies T_2 will write to some of A.

We can’t have T_1 read all of A while T_2 is writing to some of it, so S and IX are incompatible.

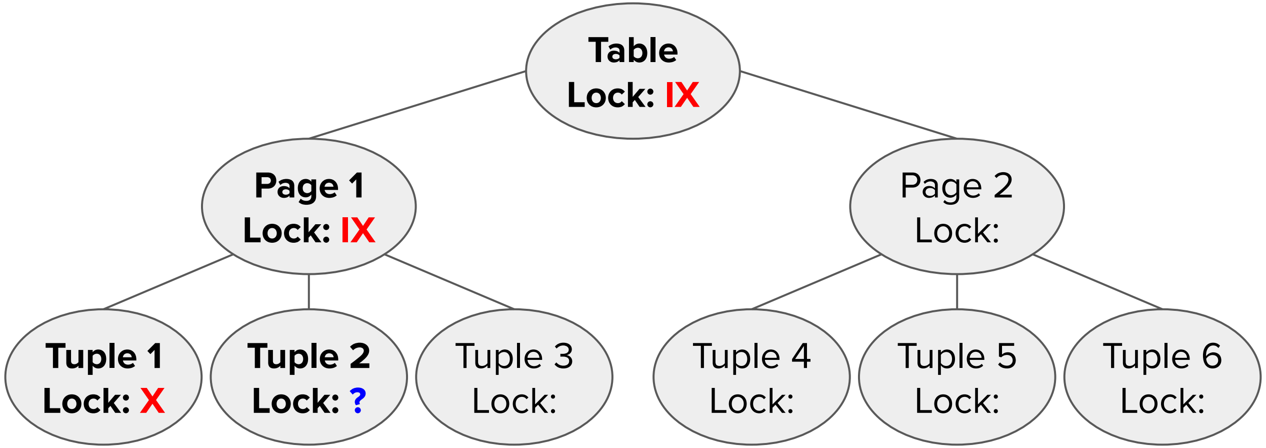
Multi-granularity Locking

(c) Consider a table which contains two pages with three tuples each, with Page 1 containing Tuples 1, 2, and 3, and Page 2 containing Tuples 4, 5, and 6.

Given that a transaction T_1 has IX(table), IX(Page 1), X(Tuple 1), which locks can be granted to a second transaction T_2 on Tuple 2?

T_1: Red

T_2: Blue



Multi-granularity Locking

(c) Consider a table which contains two pages with three tuples each, with Page 1 containing Tuples 1, 2, and 3, and Page 2 containing Tuples 4, 5, and 6.

Given that a transaction T_1 has IX(table), IX(Page 1), X(Tuple 1), which locks can be granted to a second transaction T_2 on Tuple 2?

X, S

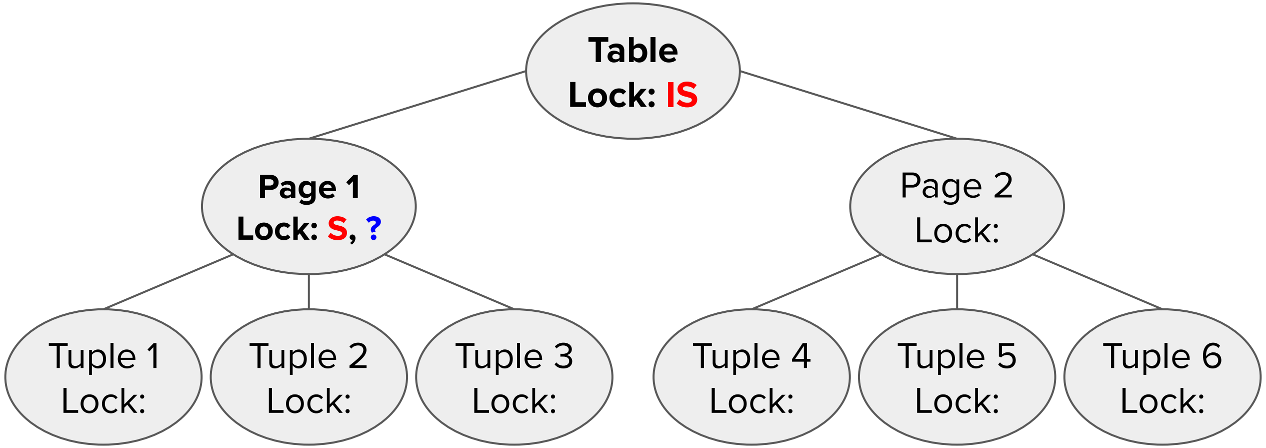
Multi-granularity Locking

(c) Consider a table which contains two pages with three tuples each, with Page 1 containing Tuples 1, 2, and 3, and Page 2 containing Tuples 4, 5, and 6.

Given that a transaction T_1 has IS(table), S(Page 1), which locks can be granted to a second transaction T_2 on Page 1?

T_1: Red

T_2: Blue



Multi-granularity Locking

(c) Consider a table which contains two pages with three tuples each, with Page 1 containing Tuples 1, 2, and 3, and Page 2 containing Tuples 4, 5, and 6.

Given that a transaction T_1 has IS(table), S(Page 1), which locks can be granted to a second transaction T_2 on Page 1?

S, IS

Attendance Link

https://cs186berkeley.net/attendance