CSE462/562: Database Systems (Spring 23)

Lecture 21: Concurrency Control

5/2/2023 & 5/4/2023

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Transactions

- Concurrent execution of user programs is essential.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the CPU busy by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A transaction is the DBMS's abstract view of a user program: a sequence of reads and writes.

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions, regardless of whether the DB is single-threaded or multi-threaded.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- Issues: Effect of interleaving transactions, and crashes.

Atomicity of transactions

- A transaction might commit after completing all its actions, or it could abort by user or system after executing some actions.
- An important property: atomicity.
  - That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS logs all actions so that it can undo the actions of aborted transactions.

ACID properties of Xact

- Atomicity
- Consistency
  - Run by itself must leave the DB in a consistent state (no IC violations)
- Isolation
  - “protected” from the effects of concurrently scheduled other transactions
- Durability
  - If a transaction has successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk.

Example, a banking database

- Consider two transactions (*Xacts*):

```
T1:    BEGIN  A=A+100,   B=B-100   END
T2:    BEGIN  A=1.06*A,  B=1.06*B  END
```

There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (cont'd)

- Consider the possible interleaving schedules

```
T1:      A=A+100,                    B=B-100
T2:                  A=1.06*A,                   B=1.06*B
```

But what about:

```
T1:      A=A+100,                                B=B-100
T2:                  A=1.06*A, B=1.06*B
```

The DBMS's view of the second schedule:

```
T1:      R(A) W(A)                                R(B) W(B)
T2:                  R(A) W(A) R(B) W(B)
```

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- *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.)

- When we discuss schedules, we only consider reads/writes/commit/abort
  - Ignores computation
- Two forms of (restricted) serializability
  - conflict serializable
  - view serializability

Anomalies with interleaved execution

- Dirty reads (WR conflict)

```
T1:     R(A), W(A),                     R(B), W(B), Abort
T2:                   R(A), W(A), C
```

- Unrepeatable reads (RW conflict)

```
T1:     R(A),                     R(A), W(A), C
T2:               R(A), W(A), C
```

Anomalies with interleaved execution

- Phantom read (RW conflict w/ predicate)

T1: R(t: P(t)) R(t: P(t)) C
T2: W(A’ , s.t. $A' \in \boldsymbol{P}$) C

- Dirty write (WW conflict)

T1: W(A) W(B) C
T2: W(A) W(B) C

Conflict serializability

- Two operations of two different transactions conflict if
  - Performed on the same object
  - At least one of them is a write

T1:	R_1 (A), W_1(A),	R_1(B), W_1(B)
T2:	R_2(A), W_2(A)	

Conflicts:

$R_1(A), W_2(A)$
$W_1(A), R_2(A)$
$W_1(A), W_2(A)$

- We can swap two adjacent nonconflicting operations without changing the final state

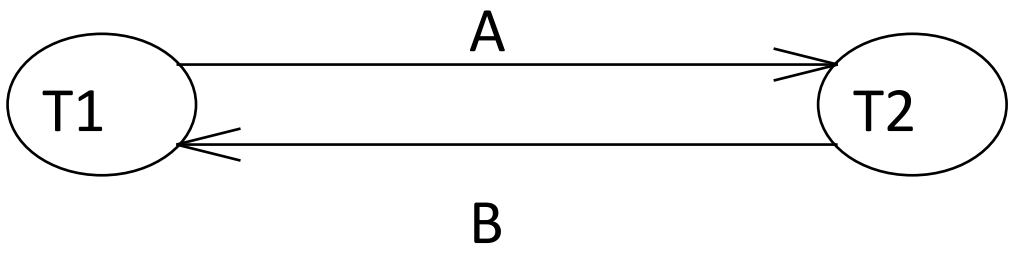
T1:	R_1 (A), W_1(A), R_1(B), W_1(B)	
T2:		R_2(A), W_2(A)

- Two schedules are conflict equivalent if one can be transformed into the other through swaps
  - Involve the same actions of the same transactions in the same order
  - Every pair of conflicting operations are ordered the same way
- Schedule S is said to be conflict serializable if it is *conflict equivalent* to some *serial* schedule S'

Determining conflict serializability

- Dependency graph
  - One node per Xact
    - edge from *Ti* to *Tj* if
      - an operation of Ti conflicts with an operation of Tj and
      - Ti's operation appears earlier in the schedule than the conflicting operation of Tj.
- Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic

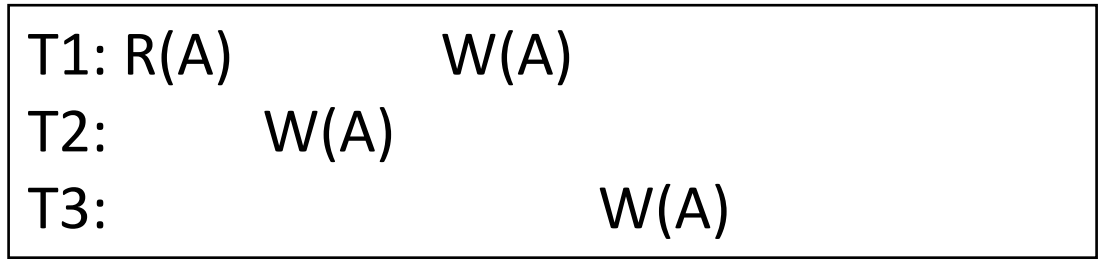
T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

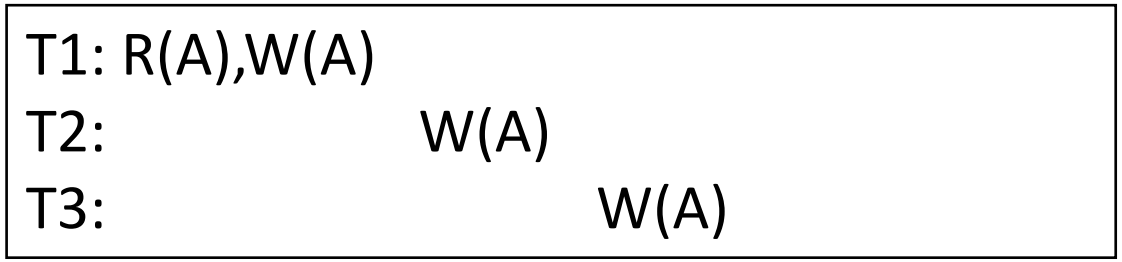


Dependency graph

View serializability

- View serializability is based on view equivalence
- Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2





View equivalent but not conflict equivalent

- View serializability is "weaker" than conflict serializability!
  - Every conflict serializable schedule is view serializable, but not vice versa!
  - I.e. admits more serializable schedules

Transaction aborts

- So far, we have not considered transaction aborts in conflict serializability
- If a transaction Ti is aborted, all its actions must be undone
  - Not only that, if Tj reads an object last written by Ti, Tj must be aborted as well!
- Many systems avoid such cascading aborts by disallowing reading an object until it is committed
  - If Ti writes an object, Tj can read this only after Ti commits.
  - Avoids non-recoverable schedules
    - where Tj reads an object previously written by Ti and Tj commits before Ti does
    - If there's a crash, the system is in a non-recoverable state
    - *Recoverable does not mean no cascading abort*
- In order to undo the actions of an aborted transaction, the DBMS maintains a log in which every write is recorded (to be discussed in more details later)
- This mechanism is also used to recover from system crashes
  - all active Xacts at the time of the crash are aborted when the system comes back up.

Pessimistic Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only conflict serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Lock Compatibility Matrix

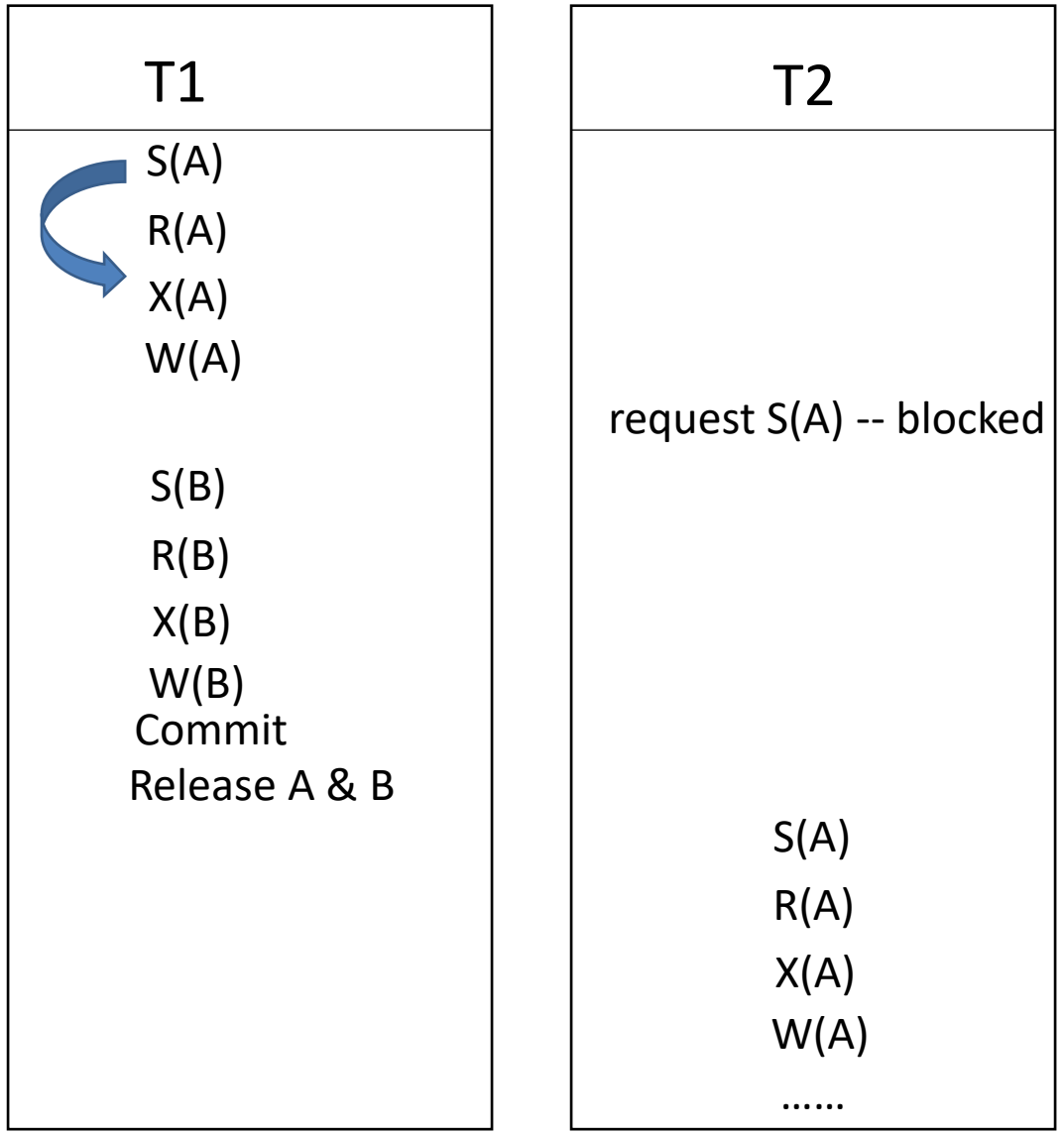
	S	X
S	√	–
X	–	–

Example: strict 2-PL

A

B

T1: A = A + 100, B = B - 100
T2: A = A - 100, B = B + 100



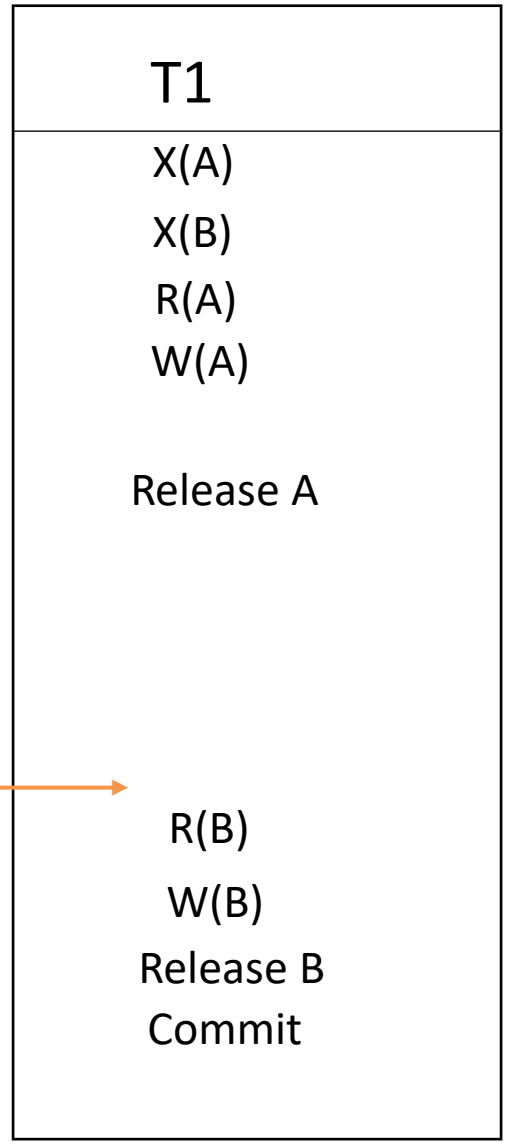
Example: non-strict 2-PL

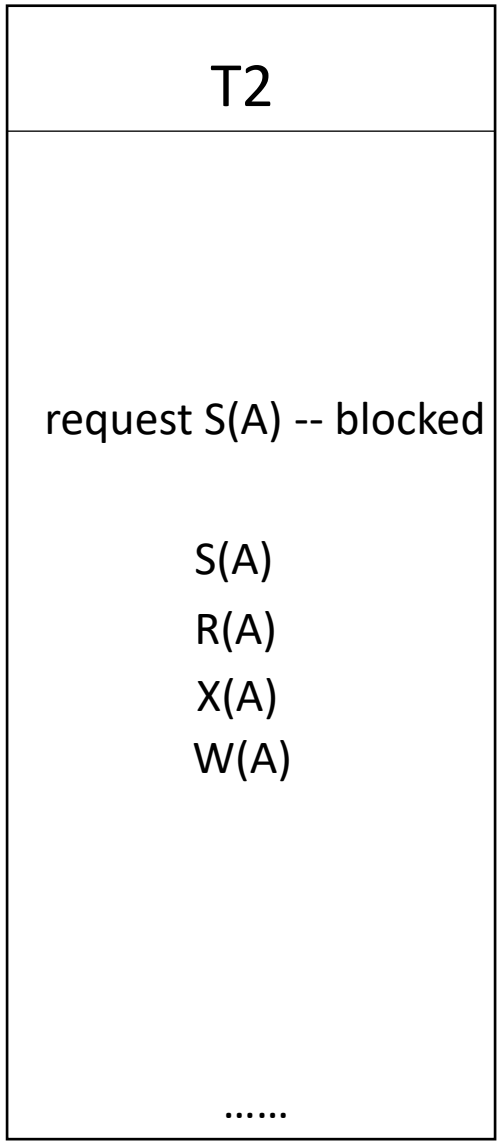
A

B

T1: A = A + 100, B = B - 100
T2: A = A - 100, B = B + 100

No new locks/lock upgrades at this point.





Example: non-strict 2-PL

A

B

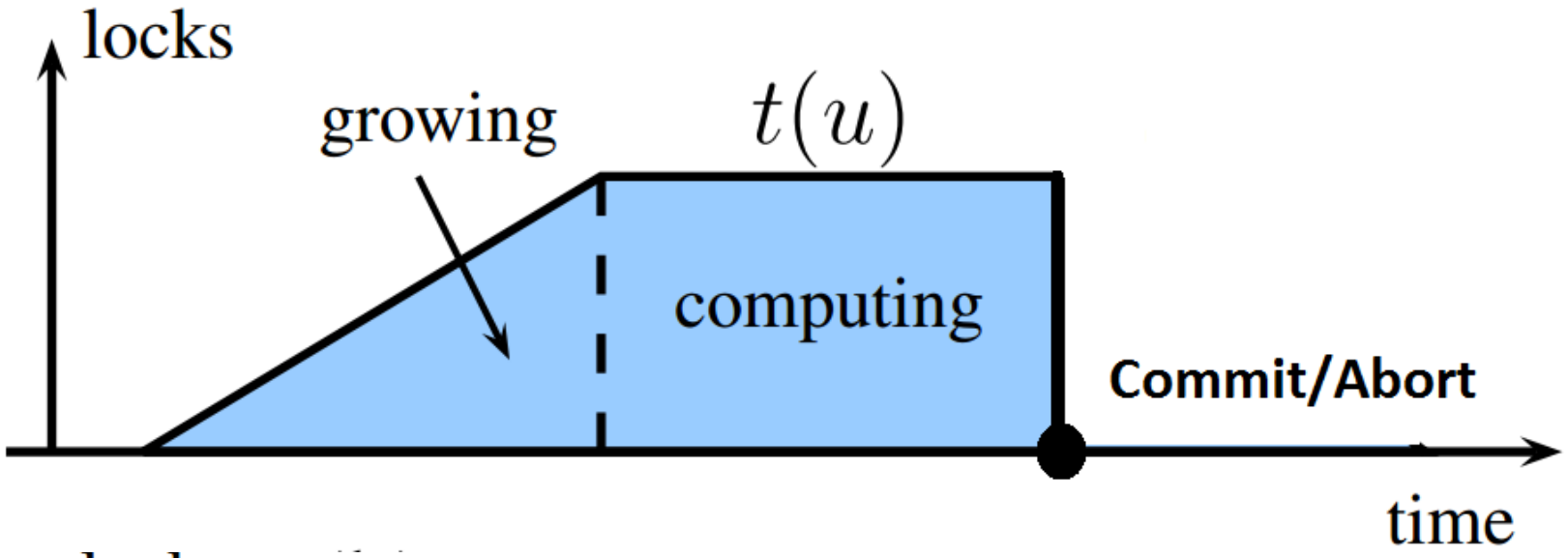
T1: A = A + 100, B = B - 100
T2: A = A - 100, B = B + 100

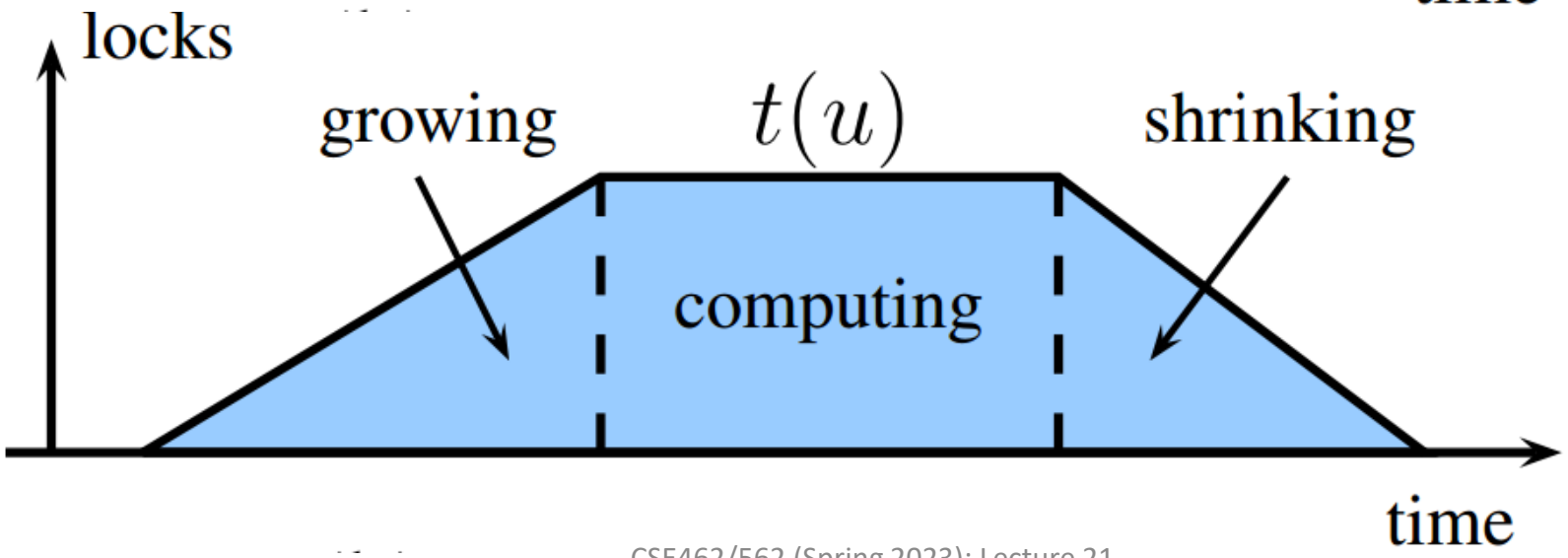
susceptible to cascading aborts!

Usually avoided in DBMS to avoid wasted work.

T1	T2
X(A)	
X(B)	
R(A)	
W(A)	
	request S(A) -- blocked
Release A	
	S(A)
	R(A)
	X(A)
	W(A)
R(B)	
W(B)	
Release B	
abort	
	abort

Strict 2-PL vs non-strict 2-PL





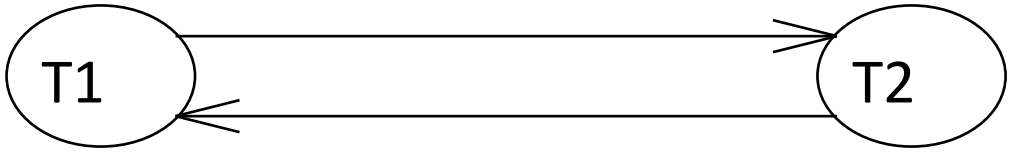
Deadlocks

A

B

T1: A = A + 100, B = B - 100
T2: B = B + 100, A = A - 100

- Create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Deadline ⇔ cycle in the wait-for graph
- Two ways to handle deadlocks
  - Deadlock prevention
  - Deadlock detection



T1	T2
S(A)	
R(A)	
X(A)	
W(A)	
	S(B)
	R(B)
	X(B)
	W(B)
S(B) -- blocked	
	S(A) -- blocked

Deadlock!

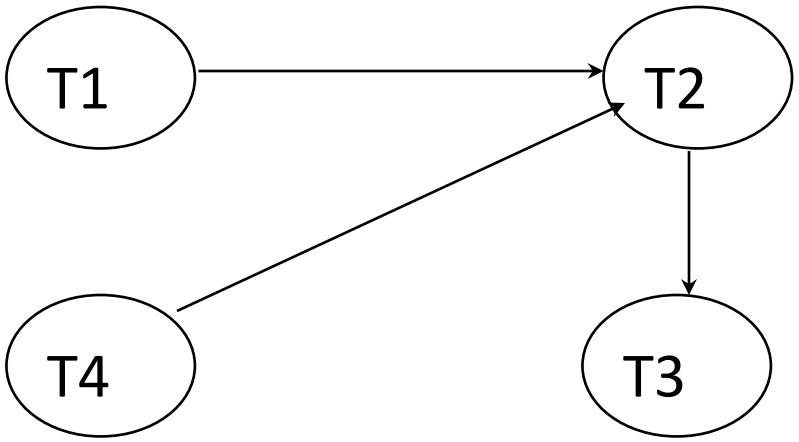
Deadlock prevention

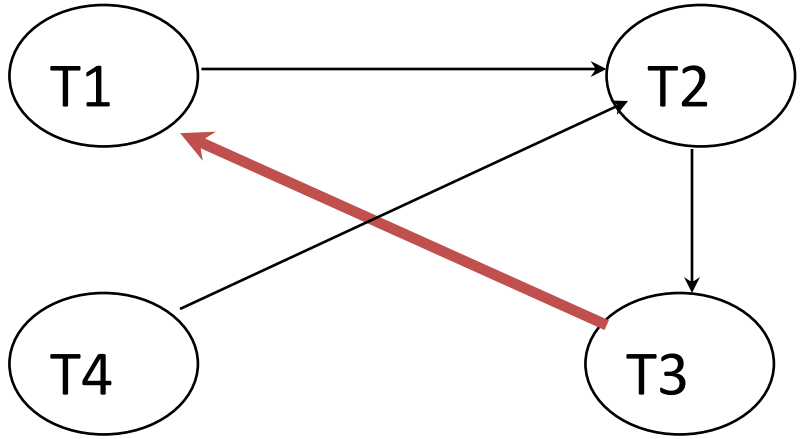
- Assign priorities based on timestamps.
  Assume Ti wants a lock that Tj holds. Two policies are possible:
  - *Wait-Die:* If Ti has lower timestamp (i.e., newer) than Tj, Ti waits; otherwise Ti aborts
  - *Wound-wait:* If Ti has lower timestamp, Tj aborts (preempted); otherwise Ti waits
- If a transaction re-starts, make sure it gets its original timestamp
  - Why? (to avoid starvation)

Deadlock detection

- Explicitly create a waits-for graph:
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- Periodically check for cycles in the waits-for graph
  - If there's a cycle, abort at least one transaction in the cycle

T1: S(A), S(D), S(B)
T2: X(B) X(C)
T3: S(D), S(C), X(A)
T4: X(B)





Deadlock detection (cont'd)

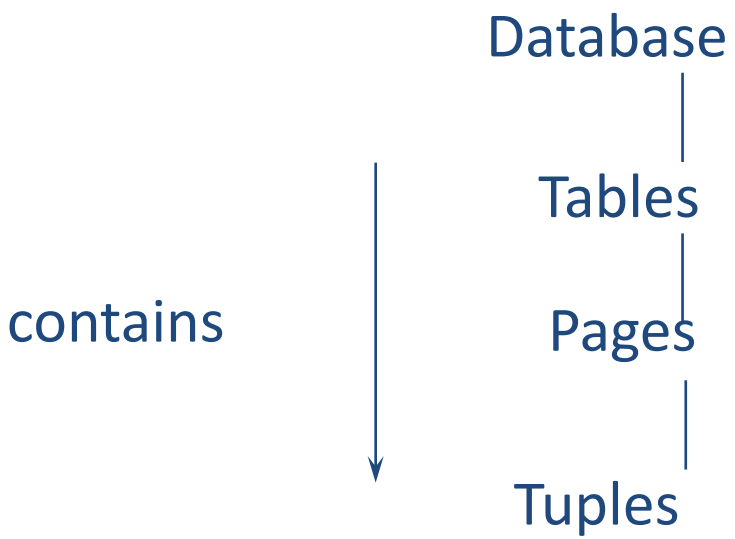
- In practice, most systems do detection
  - Experiments show that most waits-for cycles are length 2 or 3
  - Hence, only a few transactions actually need to be aborted
  - Implementations can vary
    - Can construct the graph and periodically look for cycles
      - When is the graph created ?
      - Which process checks for cycles ?
    - Can also use a "time-out" scheme
      - if T has been waiting on a lock for a long time, assume it's in a deadlock and abort

What we have glossed over

- What should we lock?
  - We assume tuples here, but that can be expensive!
  - If we do table locks, that's too conservative
  - *Multi-granularity* locking
- How to deal with phantoms?
- Locking in indexes
  - don't want to lock a B-tree root for a whole transaction!
  - more fine-grained concurrency control in indexes
- CC w/out locking (we'll omit it in this course)
  - "optimistic" concurrency control
  - "timestamp" and multi-version concurrency control
  - locking usually better, though

Multi-granularity locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to make same decision for all transactions!
- Data "containers" are nested:



Solution: new lock modes and protocols

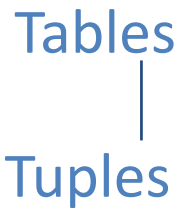
- Allow Xacts to lock at each level, but with a special protocol using new "intention" locks:
- Still need S and X locks, but before locking an item, Xact must have proper intension locks on all its ancestors in the granularity hierarchy.

- IS – Intent to get S lock(s) at finer granularity.
- IX – Intent to get X lock(s) at finer granularity.
- SIX mode: Like S & IX at the same time. Why useful?

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Example: 2-level hierarchy

- T1 scans R, and updates a few tuples:
  - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- T2 uses an index to read only part of R:
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- T3 reads all of R:
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.
- Lock escalation
  - Dynamically asks for coarser-grained locks when too many low level locks acquired



	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Dynamic Databases – The "Phantom" Problem

- If the DB is not a fixed collection of objects, even Strict 2PL (on individual items) will not assure serializability:
- Consider T1 – "Find the highest GPA among students of each age"
  - T1 locks all pages containing sailor records with *age* = 20
    - and finds the highest GPA (say, *GPA* = 3.7).
  - Next, T2 inserts a new student; *GPA* = 4.0, *age* = 20.
  - T2 also deletes student with the highest GPA (say 3.8) among those of age = 21, and commits.
  - T1 now locks all pages containing student records with age = 21, and finds highest GPA (say, *GPA* = 3.6).
- No serial execution could lead to T1's result!

The problem

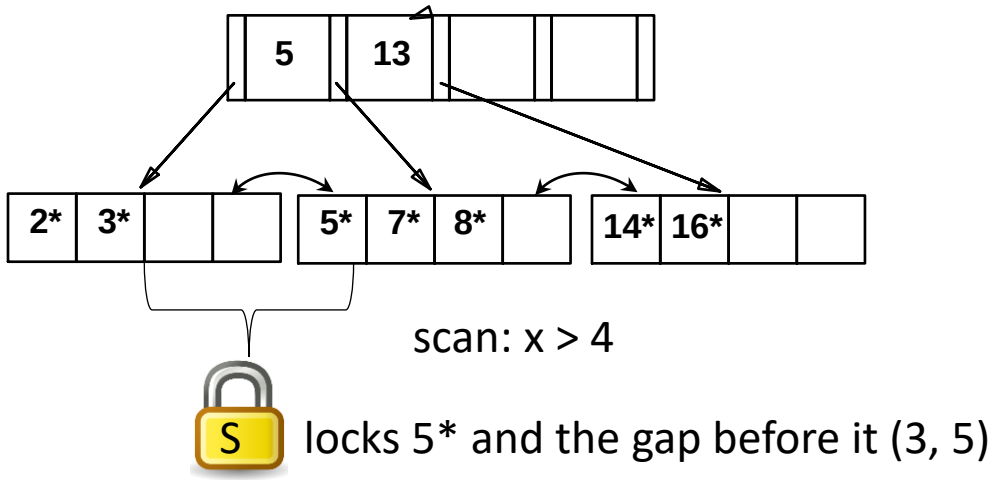
- T1 implicitly assumes that it has locked the set of all student records with *age* = 20.
  - Assumption only holds if no student records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!
  - e.g. table locks
- Solution: predicate locking

Predicate locking

- Grant lock on all records that satisfy some logical predicate, e.g. *age > 2*salary*.
- Index locking is a special case of predicate locking for which an index supports efficient implementation of the predicate lock.
  - What is the predicate in the sailor example?
- General predicate locking has a lot of locking overhead.
  - too expensive!

Instead of predicate locking

- Full table scans lock entire tables
- Range lookups do “next-key” & gap locking
  - physical stand-in for a logical range!



At this point,

insert 4: blocked
insert 10?

Lock management

- Lock and unlock requests are handled by the lock manager
- Lock table: a hash table over lock table entries
  - for various resources, e.g., records, gaps, pages, tables, …
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (S, X, IS, IX, SIX)
  - Pointer to queue of lock requests
- Locking and unlocking have to be atomic operations
  - requires *latches* (e.g. reader-writer locks/semaphores), which ensure that the process is not interrupted while managing lock table entries
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
  - Can cause deadlock problems
- Deadlock prevention/detection

Locks vs Latches

- What's common ?
  - Both used to synchronize concurrent tasks
- What's different ?
  - Locks are used for *logical consistency*
  - Latches are used for *physical consistency*
- Why treat 'em differently ?
  - Latches are short-duration lower-level locks that protects critical sections in the code
    - depends on DBMS developer to prevent deadlocks
  - Locks protects data/resources, much longer duration
    - need deadlock prevention/detection, aborting transactions using priorities
    - more lock modes, hierarchical
- Where are latches used ?
  - In a lock manager !
  - In a shared memory buffer manager
  - In a B+ Tree index
  - In a log/transaction/recovery manager

Locks vs Latches

	Latches	**Locks**
Ownership	Processes	Transactions
Duration	Very short	Long (Xact duration)
Deadlocks	No detection - code carefully !	Checked for deadlocks
Overhead	Cheap - 10s of instructions (latch is directly addressable)	Costly - 100s of instructions (have to search for lock)
Modes	S, X	S, X, IS, IX, SIX
Granularity	Flat - no hierarchy	Hierarchical

Summary

- These lectures
  - Concurrency control basics
    - Conflict serializability
    - View serializability
  - Pessimistic concurrency control
    - strict 2-phase locking
    - non-strict 2-phase locking
    - deadlock prevention and detection
    - predicate locking and next-key locking
    - lock management
      - locks vs latches
- Next lecture
  - Crash recovery