

CompSci 516

Database Systems

Lecture 19

Transactions

– Concurrency Control

Instructor: Sudeepa Roy

Announcements (Thurs, 11/31)

- HW2 deadline extended to Monday, 11/4
- Project midterm report extended to Tuesday, 11/5
- Midterm grades and exams on gradescope and sakai
 - Sample solution on sakai
 - Contact us if you have questions
- Three extra OHs for HW2 (ask questions before weekend!)
 - Tianrui, Thursday (today), 2:45 pm – 3:45pm, North 306
 - Yuchao, Thursday (today), 4:00 pm - 5:00pm, D309
 - Yuchao, Friday, 9:30 am - 10:30 am, D339
- OH By Sudeepa:
 - Friday 3-4 pm, D325

Announcements (Tues 11/5)

- Midterm report due today 11/5 on gradescope
 - One report per group
- Next week (11/12-11/14): HW3/MongoDB week!
 - You will have two Mongo Labs
 - 10% credit for HW3 in total (*not* part of 5% in-class lab credit)
 - You will have some simple tasks to finish before each lab
 - Final submission date: Tuesday 11/19
 - But try to finish as much as possible with TAs' help in labs!

Reading Material

- [RG]
 - Chapter 17.5.1, 17.5.3, 17.6
- [GUW]
 - Chapter 18.8, 18.9
 - Today's examples are from GUW (lecture slides will be sufficient for this class and exams)

Acknowledgement:

The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Today's topics

- Optimistic concurrency control (17.6.1)
- Timestamp-based concurrency control (17.6.2)
- Multi-version concurrency control (17.6.3)
- Dynamic databases and Phantom problem (17.5.1)
- Multiple—granularity locking (17.5.3)

Approaches to CC other than locking

Approaches to Concurrency Control (CC)

- Lock-based CC
 - (so far)
- Optimistic CC
 - today
- Time-stamp-based CC
 - today
- Multi-version CC
 - today



uses “timestamps” in some way

Timestamp

- Each transaction gets a unique timestamp
- e.g.
 - system's clock value when it is issued by the scheduler (assume one transactions issued on one tick of the clock)
 - or a unique number given by a counter (incremented after each transaction)

Locking is a “pessimistic or conservative” approach to CC

- Locking is a conservative approach in which **conflicts are prevented**
- Either uses “blocking” (delay) or abort
 - note the several usages of a “block”!
- Disadvantages of locking:
 - Lock management overhead
 - Deadlock detection/resolution
 - Lock contention for heavily used objects
- If only light contention for data objects, still the overhead of following a locking protocol is paid

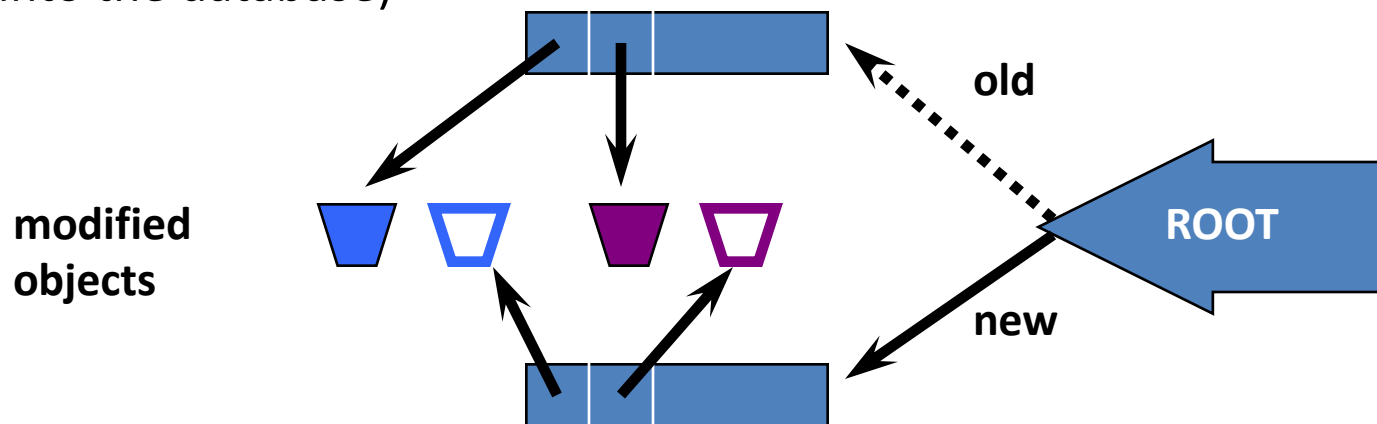
(1) Optimistic CC

Optimistic CC (or Kung-Robinson approach)

- If conflicts are rare, we might be able to gain concurrency **by not locking**, and instead **checking for conflicts before transactions commit**

Optimistic CC

- Transactions have three phases:
 1. **READ (R):** Read from the database, but make changes to “private copies” of objects (assume private workspace)
 2. **VALIDATE (V):** When decide to commit, also check for conflicts with concurrently executing transactions
 - if a possible conflict, abort, clear private workspace, restart
 3. **WRITE (W):** If no conflict, make local copies of changes public (copy them into the database)



What does Validation do?

- To validate T_2 , for each **committed** transactions T_1 such that $TS(T_1) < TS(T_2)$, one of the validation tests must be satisfied
- Validation ensures no RW, WR, WW conflicts, e.g.,
 - T1 completes all R, V, W before T2 starts
 - Or, T1 completes before W of T2 starts, and T2 does not read anything that T1 writes
 - Or, T1 completes its R before T2 starts its R, and T2 does not read/write anything that T1 writes

R = READ V = Validation W = WRITE

- Overhead:
 - Some parts have to be in “critical section” without other transactions
 - Need to maintain objects that are read/written by each transactions
 - If validation fails, need to restart, lost work

Optimistic CC vs locking

- If there are few conflicts and validation is efficient
 - optimistic CC is better than locking
- If many conflicts
 - cost of repeatedly restarting transactions hurts performance significantly

(2) Timestamp-based CC

Timestamp CC

Main Idea:

- Give each object O
 - a read-timestamp $RT(O)$, and
 - a write-timestamp $WT(O)$
 - RG uses RTS/WTS, GUW uses RT/WT, either of these is fine
 - a Commit bit $C(O)$: whether the last transaction writing O has committed
- Give each transaction T
 - a timestamp $TS(T)$ when it begins
- If
 - action a_i of T_i conflicts with action a_j of T_j ,
 - and $TS(T_i) < TS(T_j)$
- then
 - a_i must occur before a_j
- Otherwise, abort and restart violating transaction

Request for a read: $R_T(X)$

See example first
And read yourself

1. If $TS(T) \geq WT(X)$

- last written by a previous transaction -- *OK (i.e. “physically realizable”)*
- If $C(X)$ is true -- *check if previous transaction has committed*
 - Grant the read request by T
 - if $TS(T) > RT(X)$
 - set $RT(X) = TS(T)$
- If $C(X)$ is false
 - Delay T until $C(X)$ becomes true, or the transaction that wrote X aborts

2. If $TS(T) < WT(X)$

- write is not realizable -- *already written by a later trans.*
- Abort (or, Rollback) T --*i.e. abort and restart with a larger timestamp*

Request for a write: $W_T(X)$

1. If $TS(T) \geq RT(X)$ and $TS(T) \geq WT(X)$
 - last written/read by a previous transaction – *OK*
 - **Grant the write request by T**
 - write the new value of X
 - Set $WT(X) = TS(T)$
 - Set $C(X) = \text{false}$ *-- T not committed yet, set to true when T commits*
2. If $TS(T) \geq RT(X)$ but $TS(T) < WT(X)$
 - write is still realizable *--but already a later value in X*
 - If $C(X)$ is true
 - previous writer of X has committed
 - simply **ignore the write request by T**
 - but allow T to proceed without making changes to the database
 - If $C(X)$ is false
 - **Delay T** until $C(X)$ becomes true, or the transaction that wrote X aborts
- If $TS(T) < RT(X)$
 - write is not realizable *-- already read by a later transaction*
 - **Abort (or, Rollback) T**

Example

Example from GUW book

- Three transactions T1 (TS = 200), T2 (TS = 150), T3 (TS = 175)
- Three objects A, B, D
 - initially all have $RT = WT = 0$, $C = 1$ (i.e. true)
- Sequence of actions
 - $R_1(B)$, $R_2(A)$, $R_3(D)$, $W_1(B)$, $W_1(A)$, $W_2(D)$, $W_3(A)$
- Q. What is the state of the database at the end if the timestamp-based CC protocol is followed
 - i.e. report the RT, WT, C

Initial condition and Steps

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
1	R ₁ (B)					
2		R ₂ (A)				
3			R ₃ (D)			
4	W ₁ (B)					
5	W ₁ (A)					
6		W ₂ (D)				
7			W ₃ (A)			

After Step 1

WT of B is $\leq TS(T_1)$
 C = 1
 Read OK.

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 0, WT = 0, C = 1	RT = 200 , WT = 0, C = 1	RT = 0, WT = 0, C = 1
1	R ₁ (B)				RT=200	
2		R ₂ (A)				
3			R ₃ (D)			
4	W ₁ (B)					
5	W ₁ (A)					
6		W ₂ (D)				
7			W ₃ (A)			

After Step 2

WT of A is $\leq TS(T_2)$
 C = 1
 Read OK.

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 150, WT = 0, C = 1	RT = 200, WT = 0, C = 1	RT = 0, WT = 0, C = 1
1	R ₁ (B)				RT=200	
2		R ₂ (A)		RT=150		
3			R ₃ (D)			
4	W ₁ (B)					
5	W ₁ (A)					
6		W ₂ (D)				
7			W ₃ (A)			

After Step 3

WT of D is $\leq TS(T_3)$
 C = 1
 Read OK.

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 150, WT = 0, C = 1	RT = 200, WT = 0, C = 1	RT = 175, WT = 0, C = 1
1	R ₁ (B)				RT=200	
2		R ₂ (A)		RT=150		
3			R ₃ (D)			RT=175
4	W ₁ (B)					
5	W ₁ (A)					
6		W ₂ (D)				
7			W ₃ (A)			

After Step 4

WT & RT of B is $\leq TS(T_1)$
Write OK.

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 150, WT = 0, C = 1	RT = 200, WT = 200 C = 0	RT = 175, WT = 0, C = 1
1	R ₁ (B)				RT=200	
2		R ₂ (A)		RT=150		
3			R ₃ (D)			RT=175
4	W ₁ (B)				WT=200 C=0	
5	W ₁ (A)					
6		W ₂ (D)				
7			W ₃ (A)			

After Step 5

Note the change in C bit as T1 has not committed yet

RT & WT of A \leq TS(T₁)

Write ok.

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 150 WT = 200 C = 0	RT = 200 WT = 200 C = 0	RT = 175 WT = 0 C = 1
1	R ₁ (B)				RT=200	
2		R ₂ (A)		RT=150		
3			R ₃ (D)			RT=175
4	W ₁ (B)				WT=200 C=0	
5	W ₁ (A)			WT=200 C=0		
6		W ₂ (D)				
7			W ₃ (A)			

After Step 6

Object D has been read by a later transaction - abort

$RT(D) = 175 < 150 = TS(T_2)$
Abort T_2

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 150 WT = 200 C = 0	RT = 200 WT = 200 C = 0	RT = 175 WT = 0 C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(D)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(D)$ Abort				
7			$W_3(A)$			

After Step 7

$RT(A) \leq TS(T_3)$ – write ok
 $WT(A) > TS(T_3)$ and $C(A) = 0$
Delay T_3

Step	T1	T2	T3	A	B	D
	200	150	175	RT = 150 WT = 200 C = 0	RT = 200 WT = 200 C = 0	RT = 175 WT = 0 C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(D)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(D)$ Abort				
7			$W_3(A)$ Delay			

Thomas Write Rule

- If a write request comes from T on O, $TS(T) < WT(O)$, and $TS(T) \geq RT(O)$
 - violates timestamp order of T w.r.t. writer of O
 - i.e. O has been written by a later transaction T2

Thomas Write Rule:

- If $C(O) = \text{true}$, we can safely ignore such outdated writes by T
 - Otherwise "delay/block" to check whether T2 commits eventually
- no need to restart T
 - T's write is effectively followed by another write with no intervening reads
- Allows some serializable, but not conflict serializable schedules

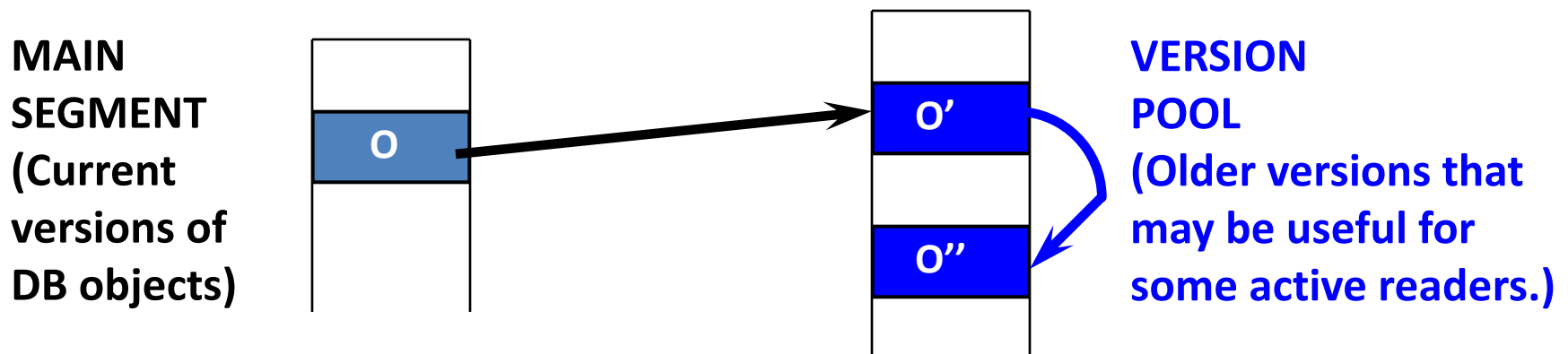
(3) Multiversion CC

Another approach to CC

- Multiversion CC
 - another way of using timestamps
 - ensures that a transaction never has to be restarted (aborted) to read an object
 - unlike timestamp-based CC
- The idea is to make several copies of each DB object
 - each copy of each object has a **write timestamp**
- Ti reads the most recent version whose timestamp precedes $TS(T_i)$

Multiversion CC

- **Idea:** Let writers make a “new” copy while readers use an appropriate “old” copy:



Readers are always allowed to proceed

- But may be “blocked” until writer commits.

Multiversion CC (Contd.)

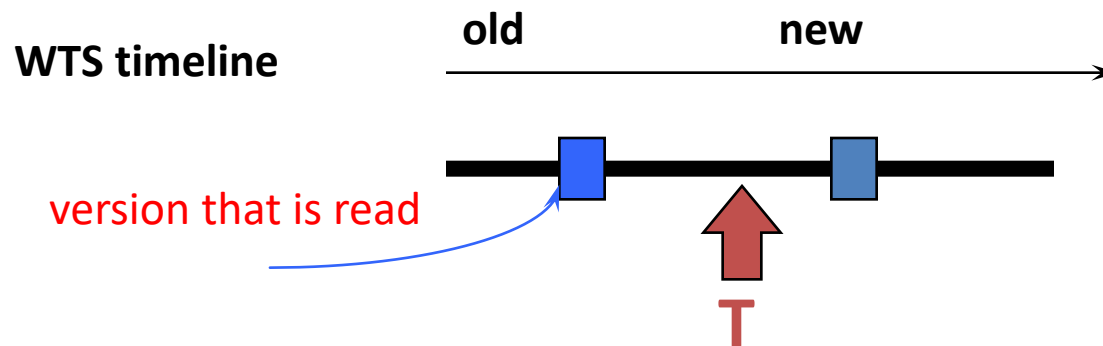
See example first
And read yourself

- Each version of an object has
 - its writer's TS as its **WT**, and
 - the timestamp of the transaction that most recently read this version as its **RT**
- Versions are chained backward
 - we can discard versions that are “too old to be of interest”
- Each transaction is classified as **Reader** or **Writer**.
 - Writer **may** write some object; Reader never will
 - Transaction declares whether it is a Reader when it begins

Reader Transaction

See example first
And read yourself

- For each object to be read:
 - Finds **newest version** with $WT < TS(T)$
 - Starts with current version in the main segment and chains backward through earlier versions
 - Update RT if necessary (i.e. if $TS(T) > RT$, then $RT = TS(T)$)
- Assuming that some version of every object exists from the beginning of time, **Reader transactions are never restarted**
 - However, might block until writer of the appropriate version commits



Writer Transaction

See example first
And read yourself

- To read an object, follows reader protocol
- To write an object:
 - must make sure that the object has not been read by a "later" transaction
 - Finds **newest version** V s.t. $WT(V) \leq TS(T)$.
- If $RT(V) \leq TS(T)$
 - T makes a copy CV of V , with a pointer to V , with $WT(CV) = TS(T)$, $RT(CV) = TS(T)$
 - Write is buffered until T commits; other transactions can see TS values but can't read version CV
- Else
 - reject write

Example

- Four transactions T1 (TS = 150), T2 (TS = 200), T3 (TS = 175), T4(TS = 225)
- One object A
 - Initial version is A_0
- Sequence of actions
 - $R_1(A), W_1(A), R_2(A), W_2(A), R_3(A), R_4(A)$
- Q. What is the state of the database at the end if the multiversion CC protocol is followed

Initial condition and Steps

A_0 existed before the transactions started

Step	T1	T2	T3	T4	A_0		
	150	200	175	225	RT=0, WT=0		
1	$R_1(A)$						
2	$W_1(A)$						
3		$R_2(A)$					
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 1

A_0 is the newest version with $WT \leq TS(T_1)$
Read A_0

Step	T1	T2	T3	T4	A_0		
	150	200	175	225	RT=0, WT=0		
1	$R_1(A)$				Read RT = 150		
2	$W_1(A)$						
3		$R_2(A)$					
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 2

- A_0 is the newest version with $WT \leq TS(T_1)$
- $RT(A_0) \leq TS(T_1)$
- Create a new version A_{150}
- Set its WT, RT to $TS(T_1) = 150$ (A_{150} named accordingly)

Step	T1	T2	T3	T4	A_0	A_{150}	
	150	200	175	225	RT=150 WT=0	RT=150 WT=150	
1	$R_1(A)$				Read RT = 150		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$					
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 3

- A_{150} is the newest version with $WT \leq TS(T_2)$
- Read A_{150}
- Update RT

Step	T1	T2	T3	T4	A_0	A_{150}	
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 4

- A_{150} is the newest version with $WT \leq TS(T_2)$
- $RT(A_{150}) \leq TS(T_2)$
- Create a new version A_{200}
- Set its WT, RT to $TS(T_2) = 200$ (A_{200} named accordingly)

Step	T1	T2	T3	T4	A_0	A_{150}	A_{200}
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	RT=200 WT=200
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					Create RT=200 WT=200
5			$R_3(A)$				
6				$R_4(A)$			

After Step 5

- A_{150} is the newest version with $WT \leq TS(T_3)$
- Read A_{150}
- DO NOT Update RT

Step	T1	T2	T3	T4	A_0	A_{150}	A_{200}
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	RT=200 WT=200
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					Create RT=200 WT=200
5			$R_3(A)$			Read	
6				$R_4(A)$			

After Step 6

- A_{200} is the newest version with $WT \leq TS(T_4)$
- Read A_{200}
- Update RT

Step	T1	T2	T3	T4	A_0	A_{150}	A_{200}
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	RT=225 WT=200
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					Create RT=200 WT=200
5			$R_3(A)$			Read	
6				$R_4(A)$			Read RT=225

Dynamic Database and Phantom Problem

Dynamic Databases

- If we relax the assumption that the DB is a fixed collection of objects
- Then even Strict 2PL will not assure serializability
- causes "Phantom Problem" in dynamic databases

Example: Phantom Problem

- T1 wants to find oldest sailors in rating levels 1 and 2

- Suppose the oldest at rating 1 has age 71
- Suppose the oldest at rating 2 has age 80
- Suppose the second oldest at rating 2 has age 63

Sailors(sid, name, age, rating)

S4, Bob, 71, 1

S7, Mary, 80, 2 Removed by T2
S3, Alice, 63, 2

- Another transaction T2 intervenes:

- **Step 1:** T1 locks all pages containing sailor records with rating = 1, and finds oldest sailor (age = 71)
- **Step 2:** Next, T2 inserts a new sailor onto a new page (rating = 1, age = 96)
- **Step 3:** T2 locks pages with rating = 2, deletes oldest sailor with rating = 2 (age = 80), commits, releases all locks
- **Step 4:** T1 now locks all pages with rating = 2, and finds oldest sailor (age = 63)

S5, Ken, 96, 1 New by T2

- No consistent DB state where T1 is “correct”

- T1 found oldest sailor with rating = 1 **before** modification by T2
- T1 found oldest sailor with rating = 2 **after** modification by T2

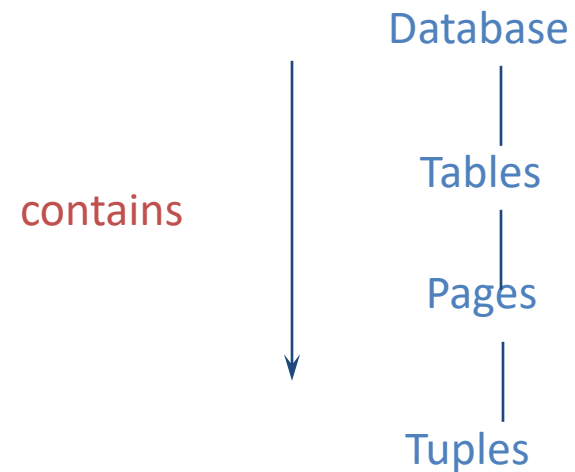
What was the problem?

- Conflict serializability guarantees serializability only if the set of objects is fixed
 - T1 implicitly and incorrectly assumed that it has locked the set of all sailor records with rating = 1
- **Solution to Phantom Problem**
 - **Index locking:** Lock the index, no new rating = 1 records can be inserted
 - **predicate locking:** Lock on “predicate” (any condition) like “rating = 1”
 - more flexible but more expensive than index locking

Multiple-granularity Locking

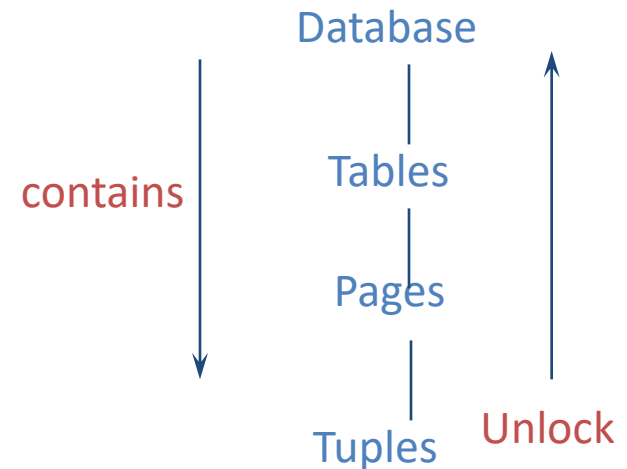
DB Objects may contain other objects

- A DB contains several files
- A file is a collection of pages
- A page is a collection of records/tuples



Carefully choose lock granularity

- If a transaction needs most of the pages
 - set a lock on the entire file, reduces locking overhead
- If only a few pages are needed
 - lock only those pages
- Need to efficiently ensure no conflicts
 - e.g. a page should not be locked by T1 if T2 already holds the lock on the file
- Acquire “intention locks” on all the ancestors before locking an item
 - Conflicts with lock requests
 - Unlock bottom-up (tuple-> pages->..)



Transaction in SQL

- SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED [;]
- BEGIN TRANSACTION
- <.... SQL STATEMENTS>
- COMMIT or ROLLBACK

- Four isolation levels : performance and serializability

	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Maybe	Maybe	Maybe
READ COMMITTED	No	Maybe	Maybe
REPEATABLE READS	No	No	Maybe
SERIALIZABLE	No	No	No

Summary

- Note the key ideas for three timestamp-based alternative approaches (to Lock-based approaches) to CC
 - Optimistic: validation tests
 - Timestamp: $RT(O)$ & $WT(O)$ on each object O
 - Multiversion: multiple versions of each object O with different WT and RT
- Note: a new action (block or delay) in addition to commit or abort
- “Phantom Problem” and why serializability/2PL fails
- New requirements and mechanisms for multiple-granularity locks