CompSci 516
Data Intensive Computing Systems

Lecture 13
Transactions

Instructor: Sudeepa Roy

Announcements

- Midterm, next Tuesday, March 1, in class
  - Everything up to Lecture 12
  - Practice problems on Friday/Saturday
- Extra office hours on Monday (also posted on Piazza)
  - Xiaodan: 10-11am, D301
  - Sudeepa: 5-6 pm, D325

Announcements

- Please submit your certification of maintaining course policy with each homework
  - the pdf is in HomeworkResources on Sakai
  - to avoid confusion or ignorance of course policies

- If you are not sure what a HW question asks for
  - you need to make sure you understand it fully by asking questions on Piazza prior to the deadline
  - Also I encourage you to have immediate notification of all Piazza posts

What will we learn?

- Last lecture:
  - Intro to transactions
- Next:
  - Concurrency Control
  - (+ some review for midterm)

Reading Material

- [RG]
  - Chapter 17.1-17.4

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

What we did in the last lecture

- Transaction
  - $R_1(A)$, $W_2(A)$, ….
  - Commit/abort
  - Lock/unlock: $S_1(A)$, $X_1(A)$, $US_1(A)$, $UX_1(A)$

- ACID properties
  - what they mean, whose responsibility to maintain each of them

- Conflicts: RW, WR, WW

- 2PL/Strict 2PL
  - all lock acquires have to precede all lock releases
  - but, they don't have to be consecutive actions on the schedule
  - exclusive locks (X) for write
  - Strict 2PL: release X locks only after commit or abort

Review: Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions
- Equivalent schedules: For any database state, the effect (on the set of objects in the database) 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 committed transactions

Review: Serializable Schedule

- If the effect on any consistent database instance is guaranteed to be identical that of "some" complete serial schedule for a set of "committed trs"
- However, no guarantee on T1-> T2 or T2 -> T1

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

serial schedule

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

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

serializable schedules

Conflict Equivalent Schedules

- Two schedules are conflict equivalent if:
  - Involve the same actions of the same transactions
  - Every pair of conflicting actions of two committed transactions is ordered the same way

- Conflicting actions:
  - both by the same transaction
    - $R_i(X)$, $W_i(Y)$
  - both on the same object by two transactions, at least one action is a write
    - $R_i(X)$, $W_j(X)$
    - $W_i(X)$, $R_j(X)$
    - $W_i(X)$, $W_j(X)$

Conflict Equivalent Schedules

- Two conflict equivalent schedules have the same effect on a database
  - all pairs of conflicting actions are in same order
  - one schedule can be obtained from the other by swapping "non-conflicting" actions
    - either on two different objects
    - or both are read on the same object

Conflict Serializable Schedules

- Schedule S is conflict serializable if S is conflict equivalent to some serial schedule

- In class:
- $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$
- to
- $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

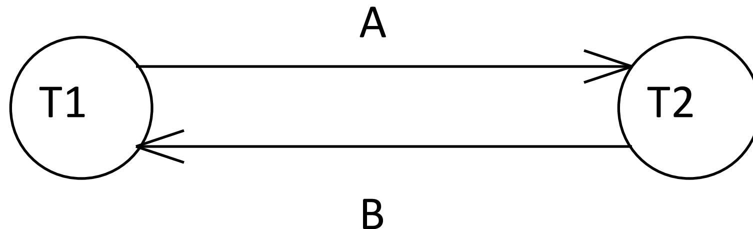
Example

- A schedule that is not conflict serializable:

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

can write it in this equivalent way as well

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



Precedence graph

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

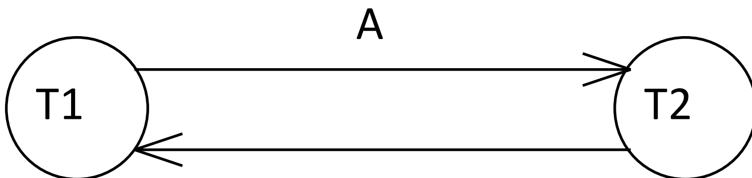
Precedence Graph

- Precedence graph:
  - Also called dependency graph, conflict graph, or serializability graph
  - One node per transaction
  - Edge from T_i to T_j if an action of T_i precedes and conflicts with one of T_j's actions
    - $W_i(A)$ --- $R_j(A)$, or
    - $R_i(A)$ --- $W_j(A)$, or
    - $W_i(A)$ --- $W_j(A)$
  - T_i must precede T_j in any serial schedule
- Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic

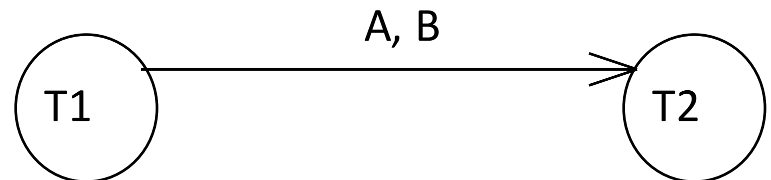
Theorem: Conflict Serializability

- Theorem: Schedule is conflict serializable if and only if its precedence graph is acyclic

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



B



$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Review: Two-Phase Locking (2PL)

- Two-Phase Locking Protocol
  - Each tr. must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - A transaction can not request additional locks once it releases any locks.
  - If a tr. holds an X lock on an object, no other tr. can get a lock (S or X) on that object.

Review: Strict 2PL

- Strict Two-phase Locking (Strict 2PL) Protocol:
  - 2PL + All locks held by a transaction are released when the transaction completes

- Strict 2PL allows only schedules whose precedence graph is acyclic
- Can never allow cycles as the X locks are being held by one tr

Strict 2PL and Conflict Serializability

- Strict 2PL allows only schedules whose precedence graph is acyclic
- Can never allow cycles as the X locks are being held by one transaction
- However, it is sufficient but not necessary for serializability
- Relaxed solution: View serializability

View 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 Tj in S1, then T1 also reads value of A written by Tj in S2
  - For all data object A, if T1 writes final value of A in S1, then T1 also writes final value of A in S2

- S is view serializable, if it is view equivalent to some serial schedule

S1

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

S2

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

More on View Serializability

- Every conflict serializable schedule is view serializable (check it yourself)
- But the converse may not be true
- If VS but not CS, would contain a "blind write" (see below)

S1

T1:	R(A)		W(A)	
T2:		W(A)		
T3:				W(A)

S2

T1:	R(A),W(A)		
T2:		W(A)	
T3:			W(A)

Lock Management

- Lock and unlock requests are handled by the lock manager
- Lock table entry:
  - Number of transactions currently holding a lock
  - Type of lock held (shared or exclusive)
  - Pointer to queue of lock requests (if the shared or exclusive lock cannot be granted immediately)
- Locking and unlocking have to be atomic operations
- Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock
- Transaction commits or aborts
  - all locks released

Deadlocks

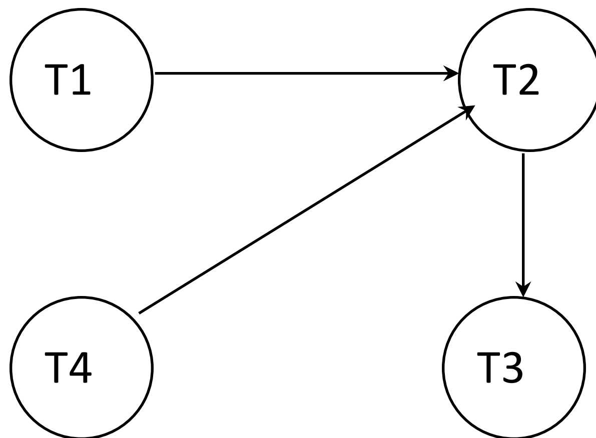
- Deadlock: Cycle of transactions waiting for locks to be released by each other.

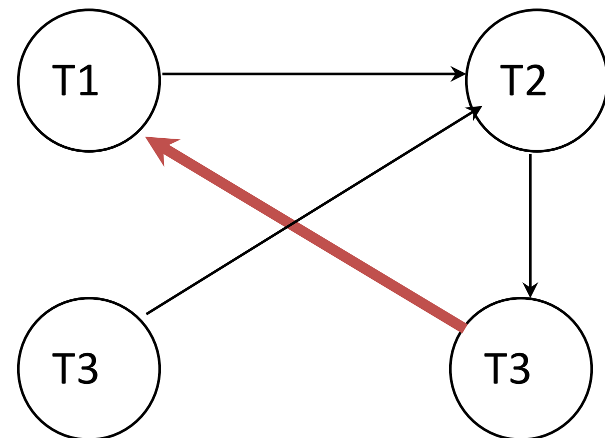
- Two ways of dealing with deadlocks:
  - Deadlock detection
  - Deadlock prevention

Deadlock Detection

Example:

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





Deadlock Detection

1. Create a waits-for graph:
   - Nodes are transactions
   - There is an edge from T1 to Tj if T1 is waiting for Tj to release a lock

- Periodically check for cycles in the waits-for graph
- Abort a transaction on a cycle and release its locks, proceed with the other transactions
  - several choices
  - one with the fewest locks
  - one has done the least work/farthest from completion
  - if being repeatedly restarted, should be favored at some point

2. Use timeout, if long delay, assume (pessimistically) a deadlock

Deadlock Prevention

- Assign priorities based on timestamps
- Assume T1 wants a lock that Tj holds. Two policies are possible:
  - Wait-Die: It T1 has higher priority, T1 waits for Tj; otherwise T1 aborts
  - Wound-wait: If T1 has higher priority, Tj aborts; otherwise T1 waits
- Convince yourself that no cycle is possible
- If a transaction re-starts, make sure it has its original timestamp
  - each transaction will be the oldest one and have the highest priority at some point

Multiple-Granularity Locks

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