CompSci 516
Data Intensive Computing Systems

Lecture 16
Transactions – Concurrency Control and Recovery

Instructor: Sudeepa Roy

Announcements

- Next Class: 3/22 after spring break

Reading Material

- [GUW]
  - Chapter 17.2.1-17.2.4 (UNDO)
  - Chapter 17.3.1-17.3.4 (REDO – next lecture)
  - Lecture material will be sufficient for exams and assignments

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

Last Lecture

- Dynamic Database + Phantom Problem
- Multiple-granularity locks
- Alternatives timestamp-based (not 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

Today

CC

- Algorithms and examples for CC approaches in the last lecture
  - wrap up for CC

Recovery

- UNDO log
- REDO log
  - to be continued in the next 1-2 lecture
  - up to slide#60 today

Algorithms and Examples of CC

skipping optimistic CC – where you have to check the intersection of READSET and WRITESET of pairs of transactions, and overlaps of three phases (read-validation-write) using Tests 1, 2, 3

Input and Possible actions

- Request comes for $R_T(X)$ or $W_T(X)$
  - transaction T wants to read or write object X

- Possible actions
  1. Grant the request
  2. Abort T and restart with a new timestamp
     - also called Rollback
  3. Delay (block) T and decide later whether to grant the request or abort
  4. Ignore the request

Notations

- TS(T)
  - unique timestamp for transaction T
- RT(X)
  - the read time of X
    = the highest timestamp of a transaction that has read X
- WT(X)
  - the write time of X
    = the highest timestamp of a transaction that has written X
- C(X)
  - the commit bit of X
  - either true or false
  - true if and only if the most recent transaction to write X has already committed
  - avoids dirty read of uncommitted data and makes the schedule recoverable

Algorithm:
Timestamp-based scheduling

Request for a read: $R_T(X)$

1. If TS(T) >= WT(X)
   - last written by a previous transaction –- *OK (i.e. "physically realizable")*
   - If C(X) is true
     - 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 *--written by a later transaction*
   - Abort (or, Rollback) T *--i.e. abort and restart with a larger timestamp*

Request for a write: $W_T(X)$

1. If TS(T) >= RT(X) and TS(T) >= 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) = false *// T not committed yet*
2. If TS(T) >= 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 1: Timestamp-based scheduling

Example

- Three transactions T1 (TS = 200), T2 (TS = 150), T3 (TS = 175)
- Three objects A, B, C
  - initially all have RT = WT = 0, C = 1 (i.e. true)
- Sequence of actions
  - $R_1(B), R_2(A), R_3(C), W_1(B), W_1(A), W_2(C), W_3(A)$
- 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	C
	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			$R_3(C)$			
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 1

WT of B is <= TS(T_1)
C = 1
Read OK.

Step	T1	T2	T3	A	B	C
	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			$R_3(C)$			
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 2

WT of A is <= TS(T_2)
C = 1
Read OK.

Step	T1	T2	T3	A	B	C
	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			$R_3(C)$			
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 3

WT of C is <= TS(T_3)
C = 1
Read OK.

Step	T1	T2	T3	A	B	C
	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			$R_3(C)$			RT=175
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 4

WT & RT of B is <= TS(T_1)
Write OK.

Step	T1	T2	T3	A	B	C
	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			$R_3(C)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 5

RT & WT of A <= TS(T_1)
Write ok.

Step	T1	T2	T3	A	B	C
	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			$R_3(C)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(C)$				
7			$W_3(A)$			

After Step 6

$RT(C) = 175 < 150 = TS(T_2)$

Abort T_2

Step	T1	T2	T3	A	B	C
	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			$R_3(C)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(C)$ Abort				
7			$W_3(A)$			

After Step 7

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

Step	T1	T2	T3	A	B	C
	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			$R_3(C)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(C)$ Abort				
7			$W_3(A)$ Delay			

Example 2: Multiversion CC

Algorithm summary (see Lec 15)

- Read request:
  - Finds newest version with WT <= TS(T)
  - Update RT if necessary (i.e. if TS(T) > RT, then RT = TS(T))
- Write request:
  - Finds newest version V s.t. WT <= TS(T).
  - If RT(V) <= TS(T)
    - T makes a new copy CV of V, with WT(CV) = TS(T), RT(CV) = TS(T)
    - simplified – we will not consider whether T committed or not before other transaction reads this copy in Multiversion CC – i.e. assume that a copy can be read right after it has been created
  - Else
    - reject write - abort

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 <= 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 <= $TS(T_1)$
- $RT(A_0) < TS(T_1)$
- 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 <= 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 <= $TS(T_2)$
- $RT(A_{150})$ <= $TS(T_2)$
- 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 <= 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 <= 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

Transaction Support in SQL-92

- SET TRANSACTION READ ONLY;
- SET TRANSACTION READ WRITE;
- SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
- ……

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Transaction Recovery and Logs

Review: The ACID properties

- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of other transactions.
- **D** urability: If a transaction commits, its effects persist.

- Which property did we cover in CC?

Review: The ACID properties

- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of other transactions.
- **D** urability: If a transaction commits, its effects persist.

- Which property did we cover in CC? : Isolation

Review: The ACID properties

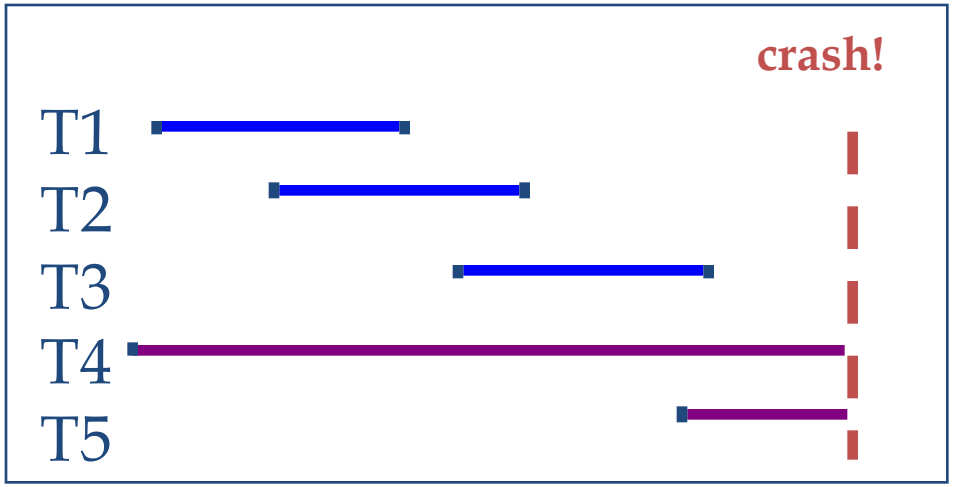
- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of other transactions.
- **D** urability: If a transaction commits, its effects persist.

- Next: The Recovery Manager guarantees Atomicity & Durability.
- Recall that Consistency is programmer's responsibility

Motivation: A & D

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running? (Causes?)

❖ Desired Behavior after system restarts:
- T1, T2 & T3 should be durable.
- T4 & T5 should be aborted (effects not seen).



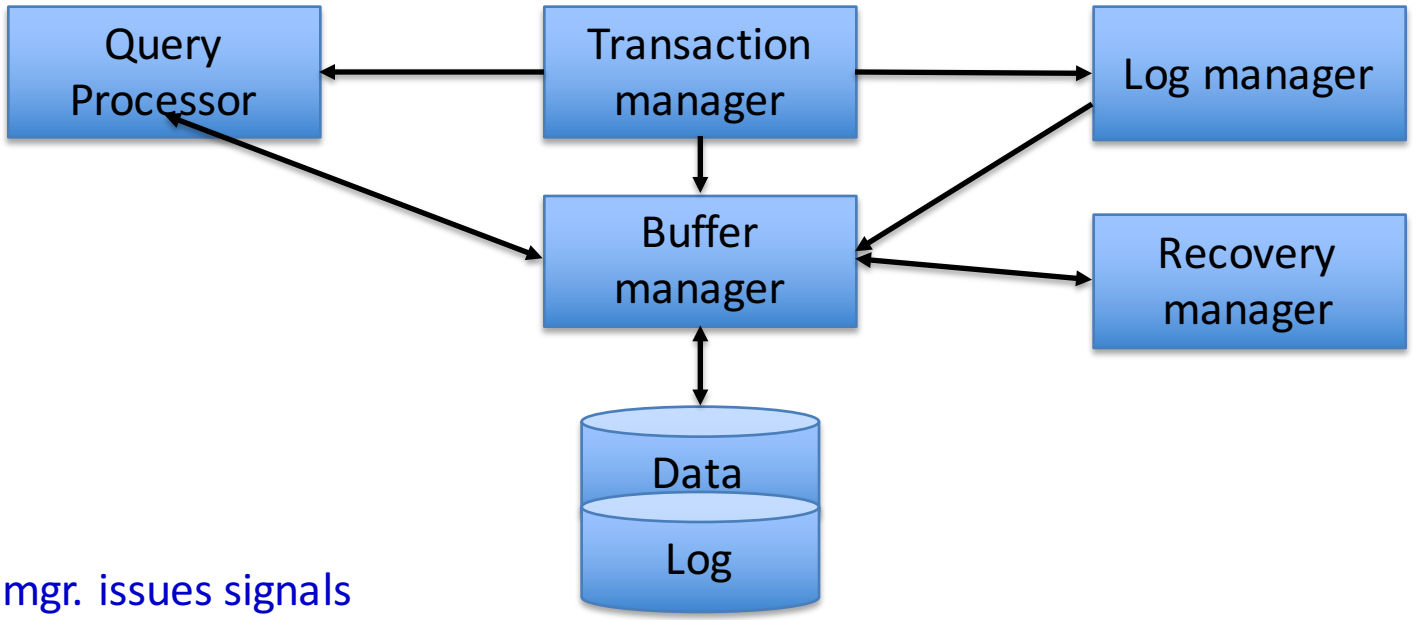
Recovery: A & D

- Atomicity
  - by ”undo”ing actions of transactions that did not commit
- Durability
  - by making sure that all actions of committed transactions survive crashes and system failure

Reasons for failure

- Erroneous data entry
  - caught by constraints and triggers
- Media failure
  - disk crashes
- System failure
  - power failure and state of the transaction is lost
- Catastrophic failure
  - explosions, fire, vandalism

DB Architecture for Transactions



- Tr. mgr. issues signals
  - to log mgr. to store log records
  - to buffer mgr. about when it should copy buffer to disk
  - to query processor to execute queries/operations that compromise the transaction
- Log mgr.
  - maintains log, deals with buffer mgr. since first log appears in buffer then is written to disk
- Recovery mgr.
  - If there is a crash, it is activated
  - examines the log and repairs data if necessary
- Note: access to the disk is through the buffer mgr.

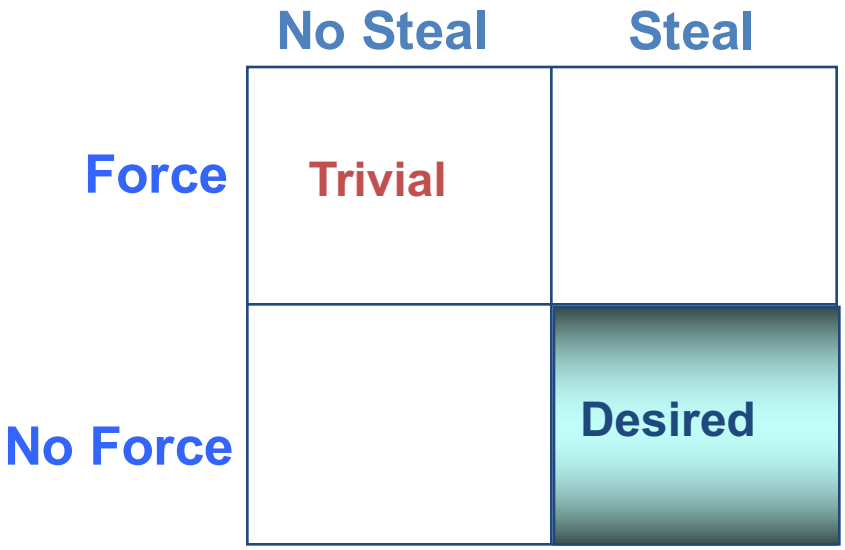
Assumptions

- Concurrency control is in effect
- Updates are happening "in place".
  - i.e. data is overwritten on (deleted from) the disk.
- What is a simple scheme to guarantee Atomicity & Durability?

Handling the Buffer Pool

- Force every write to disk?
  - Poor response time.
  - But provides durability.

- Steal buffer-pool frames from uncommited transactions?
  - If not, poor throughput.
  - If so, how can we ensure atomicity?



More on Steal and Force

- STEAL (why enforcing Atomicity is hard)
  - To steal frame F: Current page in F (say P) is written to disk; some transaction holds lock on P
    - What if the transaction with the lock on P aborts?
    - Must remember the old value of P at steal time (to support UNDOing the write to page P)

- NO FORCE (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

Basic Idea: Logging

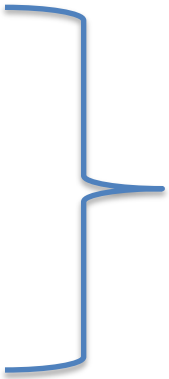
- Record REDO and UNDO information, for every update, in a log.
  - Sequential writes to log (put it on a separate disk).
  - Minimal info (diff) written to log, so multiple updates fit in a single log page.

- Log: An ordered list of REDO/UNDO actions
  - Log record may contain:

    <Tr.ID, pageID, offset, length, old data, new data>

Different types of logs

- UNDO
- REDO
- UNDO/REDO



GUW 17.2, 17.3, 17.4
(Lecture material will be sufficient for HWs and Exams)

- ARIES

Next lecture
RG 18
and a research paper

Slides on UNDO logging covered in this lecture (Lecture 16)
have been moved to Lecture 17