CS 6400 A

Database Systems Concepts and Design

Lecture 19

11/03/25

Announcements

- Project Milestone due tonight

- This is a quite week otherwise

Desirable Properties of Transactions: ACID

- Atomicity: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.

- Consistency: A correct execution of the transaction must take the database from one consistent state to another.

- Isolation: A transaction should not make its updates visible to other transactions until it is committed.

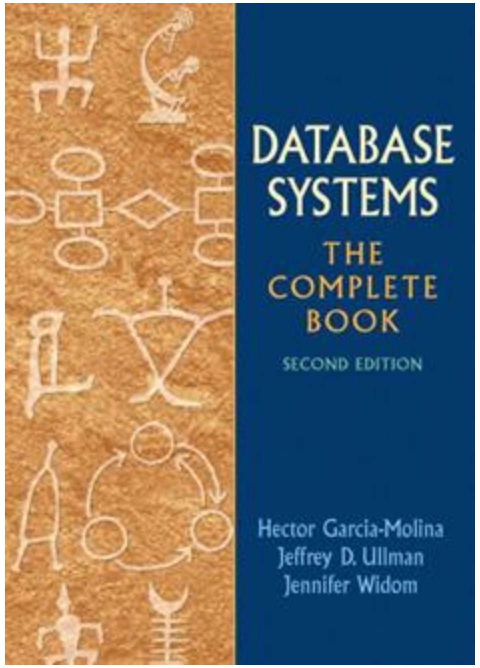
- Durability: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

This class: ensuring consistency & isolation via concurrency control

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 18 – Concurrency Control



Supplementary materials

Fundamental of Database Systems (7th Edition)

- Chapter 21 - Concurrency Control Techniques

Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang.

Agenda

1. Schedule

2. Lock-based Concurrency Control

3. Optimistic Concurrency Control

1. Schedule

Schedule

A transaction is seen by DBMS as a list of actions.

- READ, WRITE of database objects
- ABORT, COMMIT

Assumption: Transactions communicate only through READ and WRITE

Schedule is a list of actions from a set of transactions as seen by the DBMS

- Two actions from *the same transaction T* MUST appear in the schedule in the same order that they appear in T
- Intuitively, a schedule represents an actual or potential execution sequence

Transaction primitives

- INPUT(X): copy block X from disk to memory
- READ(X, t): copy X to transaction's local variable t
  (run INPUT(X) if X is not in memory)
- WRITE(X, t): copy value of t to X (run INPUT(X) if X is not in memory)
- OUTPUT(X): copy X from memory to disk

Schedule

- Actions taken by one or more transactions

T1	*T2*
READ(*A*, *t*)	READ(*A*, s)
t := *t*+100	*s* := *s**2
WRITE(*A*, *t*)	WRITE(*A*, *s*)
READ(*B*, *t*)	READ(*B*, *s*)
t := *t*+100	*s* := *s**2
WRITE(*B*, *t*)	WRITE(*B*, *s*)

Characterizing Schedules based on Serializability (1)

Serial schedule

- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
  - Basically, actions from different transactions are NOT interleaved
  - Otherwise, the schedule is called nonserial schedule.

Serializable schedule

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

Serial and serializable schedules are guaranteed to preserve the consistency of database states

Serial schedule

- One transaction is executed at a time

T1	*T2*	*A*	*B*
		25	25
READ(*A*, *t*)			
t := *t*+100			
WRITE(*A*, *t*)		125	
READ(*B*, *t*)			
t := *t*+100			
WRITE(*B*, *t*)			125
	READ(*A*, s)		
	s := *s**2		
	WRITE(*A*, *s*)	250	
	READ(*B*, *s*)		
	s := *s**2		
	WRITE(*B*, *s*)		250

Schedule: (T1, T2)

Q: Do serial schedules allow for high throughput?

Serializable schedule

- There exists a serial schedule with the same effect

T1	*T2*	*A*	*B*
		25	25
READ(*A*, *t*)			
t := *t*+100			
WRITE(*A*, *t*)		125	
	READ(*A*, s)		
	s := *s**2		
	WRITE(*A*, *s*)	250	
READ(*B*, *t*)			
t := *t*+100			
WRITE(*B*, *t*)			125
	READ(*B*, *s*)		
	s := *s**2		
	WRITE(*B*, *s*)		250

Same effect as (T1, T2)

Serializable schedule

- This is not serializable (values for A, B changed)

T1	*T2*	*A*	*B*
		25	25
READ(*A*, *t*)			
t := *t*+100			
WRITE(*A*, *t*)		125	
	READ(*A*, s)		
	s := *s**2		
	WRITE(*A*, *s*)	250	
	READ(*B*, *s*)		
	s := *s**2		
	WRITE(*B*, *s*)		50
READ(*B*, *t*)			
t := *t*+100			
WRITE(*B*, *t*)			150

Q: Is this schedule serializable?

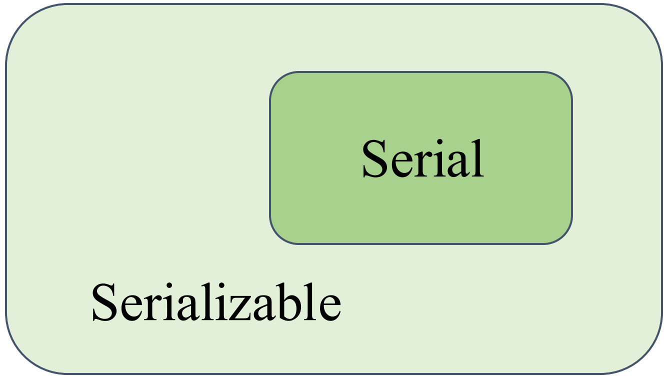
Serializable schedule

- Serializable, but only due to the detailed transaction behavior

T1	*T2*	*A*	*B*
		25	25
READ(*A*, *t*)			
t := *t*+100			
WRITE(*A*, *t*)		125	
	READ(*A*, s)		
	s := *s*+200		
	WRITE(*A*, *s*)	325	
	READ(*B*, *s*)		
	s := *s*+200		
	WRITE(*B*, *s*)		225
READ(*B*, *t*)			
t := *t*+100			
WRITE(*B*, *t*)			325

Same effect as (T1, T2)

Serial vs Serializable Schedule



Being serializable is not the same as being serial

Being serializable implies that the schedule is a correct schedule.

- It will leave the database in a consistent state.

Interleaving improves efficiency due to concurrent execution, e.g.,

- While one transaction is blocked on I/O, the CPU can process another transaction
- Interleaving short and long transactions might allow the short transaction to finish sooner (otherwise it need to wait until the long transaction is done)

Interleaving & Isolation

The DBMS has freedom to interleave TXNs

However, it must pick an interleaving or **schedule** such that isolation and consistency are maintained

- Must be *as if* the TXNs had executed serially!

A<u>CI</u>D

DBMS must pick a schedule which maintains isolation & consistency

Abstract view of TXNs: reads and writes

Serializability is hard to check - cannot always know detailed behaviors

DBMS's abstract view of transactions:

$r_i(X)$: Ti reads X
$w_i(X)$: Ti writes X

T1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Serializable schedule: $r_1(A)$; $w_1(A)$; $r_2(A)$; $w_2(A)$; $r_1(B)$; $w_1(B)$; $r_2(B)$; $w_2(B)$;

Conflicts: Anomalies with Interleaved Execution

Conditions for conflicts:

- The operations must belong to different transactions (no conflict within the same transaction).
- The operations must access the same database object
- At least one of the operations must be a write operation.

Types of conflicts:

- Write-Read (WR)
- Read-Write (RW)
- Write-Write (WW)

Implication for schedules:
A pair of consecutive actions that cannot be interchanged without changing behavior

DB isolation levels define which types of conflicts a database will prevent or allow.

WR Conflict

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

Reading Uncommitted Data (WR Conflicts, “dirty reads”):

- transaction T2 reads an object that has been modified by T1 but not yet committed

RW Conflict

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

Unrepeatable Reads (RW Conflicts):

- T2 changes the value of an object A that has been read by transaction T1, which is still in progress
- If T1 tries to read A again, it will get a different result

WW Conflict

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

Overwriting Uncommitted Data (WW Conflicts, “lost update”):

- T2 overwrites the value of A, which has been modified by T1, still in progress
- Suppose we need the salaries of two employees (A and B) to be the same
  - T1 sets them to $1000
  - T2 sets them to $2000

Characterizing Schedules based on Serializability (2)

Conflict equivalent

- 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

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S'.

Conflict-serializable schedule

The schedule respects the internal ordering of each transaction

- Conflict-equivalent to serial schedule

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

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

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

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

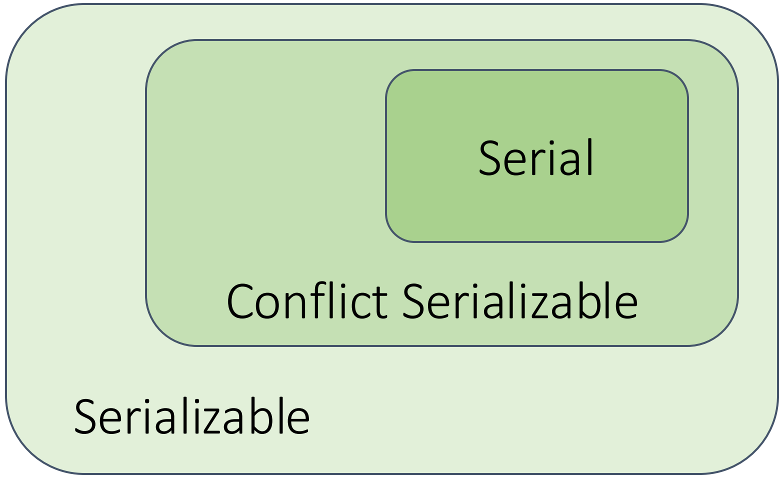
Serial $r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B);$

Conflict-serializable schedule

- A conflict-serializable schedule is always serializable
- But not vice versa (e.g., serializable schedule due to detailed transaction behavior)

S1: $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$ — Serial

S2: $w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$ — Serializable, but not conflict serializable



In-class Exercise

- Are there conflict-equivalent schedules to (T1, T2) that interleaves the two transactions?

T1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

Testing for conflict serializability

Through a precedence graph:

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

Precedence graph

Can use to decide conflict serializability

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

** Also called dependency graph, conflict graph, or serializability graph*

Precedence graph

Can use to decide conflict serializability

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$ T1 → T2 → T3

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$ T1 T2 T3

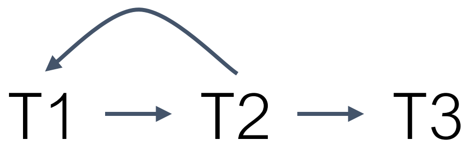
- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
  - Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

Precedence graph

Can use to decide conflict serializability

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$



- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
  - Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

Precedence graph

Can use to decide conflict serializability

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

This is conflict serializable

T1 → T2 → T3

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

This is not because of cycle

T1 → T2 → T3

- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
  - Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

In-class Exercise

- What is the precedence graph for the schedule:

$r_1(A); r_2(A); r_1(B); r_2(B); r_3(A); r_4(B); w_1(A); w_2(B);$

- One node per committed transaction
- Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
  - Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)