Database Systems

COMP 3010E FALL 2025

LECTURE 14 CONCURRENCY CONTROL

Agenda

- Transaction
- Concurrency Control
  - Lock-Based Concurrency Control Protocols
  - Timestamp-Based Concurrency Control Protocols

Transaction

A transaction is a sequence of actions on the database, that is:

<u>Example</u>: (Transaction T) Transferring 5000$ from account A to account B.

```
read(A);
A := A – 5000;
write(A);
read(B);
B := B + 5000;
write(B);
Commit;
```

Notes:

- **read**(X) transfers data item X from the database to the transaction's workspace; **write**(X) writes data item X from the workspace back to the database
- Transaction **Commit**: The transaction is completed successfully, and the database enters a new state.
- Transaction **Rollback**: The transaction is aborted, and all changes are undone, restoring the database to its previous state.

Transaction Management

A transaction is a sequence of actions on the database, that is.

Atomic
- Transaction cannot be subdivided

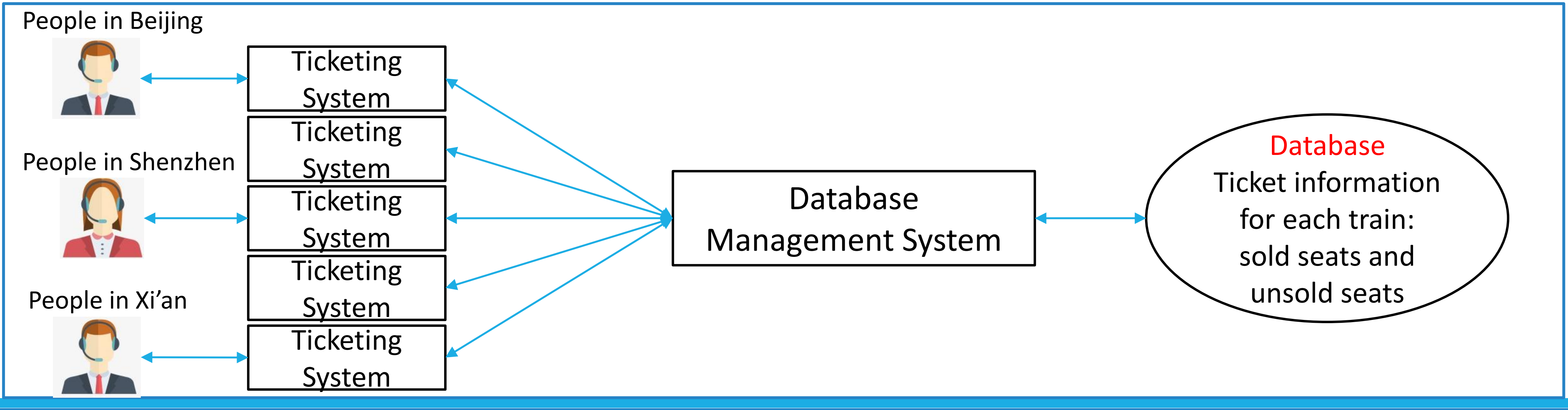
Consistent
- Transaction must take the database from one valid state to another
- e.g., database is consistent with the integrity rules before and after the transaction

Isolated
- One transaction must not interfere with another

Durable
- Database changes are permanent. Committed transactions must remain permanent.

Concurrency Control

- **Why is concurrency control necessary** - Multiple users executing transactions simultaneously on the database ensures no conflicts or data inconsistencies occur
  - Train ticketing systems: prevent overbooking and seat conflicts; ensure correct handling of ticket purchases and refunds, and accurate pricing.
  - Online shopping platforms, course selection systems.
  - Concurrency can improve system throughput and reduce waiting time.



Transaction Scheduling and Serializability

- **<u>Transaction scheduling</u>**: The execution order of the basic steps (read, write, and other control operations such as locking and unlocking) of a group of transactions is called a schedule for that group of transactions.
- **<u>Concurrent (or parallel) scheduling</u>**: Executing the operations in a group of concurrently running transactions in a certain order.

S	T_1	T_2
1	Read A	
2	A=A-10	
3	Write A	
4	Read B	
5	B=B+10	
6	Write B	
7		Read B
8		B=B-20
9		Write B
10		Read C
11		C=C+20
		Write C

Serial scheduling

S	T_1	T_2
1	Read A	
2		Read B
3	A=A-10	
4		B=B-20
5	Write A	
6		Write B
7	Read B	
8		Read C
9	B=B+10	
10		C=C+20
11	Write B	
12		Write C

S	T_1	T_2
1	Read A	
2	A=A-10	
3		Read B
4	Write A	
5		B=B-20
6	Read B	
7		Write B
8	B=B+10	
9		Read C
10	Write B	
11		C=C+20
		Write C

Concurrent scheduling

Transaction Scheduling and Serializability

- A Simple Notation Model for Transaction Scheduling
  - $r_T(A)$: Transaction T reads database object A.
  - $w_T(A)$: Transaction T writes to database object A.

- Here, A and B represent database objects, such as tuple values or attribute values.
- This model focuses solely on the read and write operations; other computational steps that may occur in memory between these operations are not represented.

Transaction Scheduling and Serializability

- **Concurrent Schedules**: A schedule is correct if and only if the resulting database state produced by the concurrent schedule is identical to the state produced by running those transactions serially in some order.

- **Serializability**: A schedule is considered serializable (or possesses the property of serializability) if its effect on the final state of the database is equivalent to the effect of some serial schedule, regardless of the initial state of the database.

Transaction Scheduling and Serializability

- **Conflict**: Two consecutive operations in a schedule, from different transactions, that access the same data object and at least one of them is a write operation.

  - Conflicting operations are non-commutative. Swapping their order may alter the final outcome of the transactions.

    A = 100. T1: Read(A), T2: Write(A=0), T1: print(A)

    T2: Write(A=0), T1: Read(A), T1: print(A)

  - Non-conflicting operations are commutative. Swapping their order does not change the outcome of the schedule.

Transaction Scheduling and Serializability

- **Conflict**: Two consecutive operations in a schedule, from different transactions, that access the same data object and at least one of them is a write operation.

  - Two write operations on the same data object by different transactions conflict (Write-Write conflict).

  - A read and a write operation on the same data object by different transactions conflict (Read-Write or Write-Read conflict).

Transaction Scheduling and Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are conflict equivalent.
- We say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

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

- Two objects: A and B

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

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

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

$\Downarrow$

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

Lock-Based Concurrency Control Protocols

- **Lock**: A lock is a mechanism used to control concurrency. It is a global variable that represents the "permission" to perform specific operations on a particular element.

- Before a transaction operates on an element, it must obtain the corresponding lock.

- If the element is already locked by another transaction and cannot be re-locked, the transaction must wait for the other transaction to release the lock.

- Once the operation is completed, the lock must be released. After the transaction ends, all locks held by it are released.

Example (Lock)

The scheduler can use locks to achieve (but does not guarantee to achieve) conflict serializability.

T1	T2
L_1(A); READ(A, t)	
t := t+100	
WRITE(A, t);	
U_1(A)	
L_1(B)	
	L_2(A); READ(A,s)
	s := s*2; WRITE(A,s);
	U_2(A);
	L_2(B); “reject…”
READ(B, t)	
t := t+100	
WRITE(B,t);	
U_1(B);	
	obtain locks
	READ(B,s)
	s := s*2; WRITE(B,s);
	U_2(B);

T1	T2
L_1(A); READ(A, t)	
t := t+100	
WRITE(A, t);	
U_1(A)	
L_1(B)	
READ(B, t) t := t+100	
WRITE(B,t);	
U_1(B);	
	L_2(A); READ(A,s)
	s := s*2; WRITE(A,s);
	U_2(A);
	L_2(B);
	READ(B,s)
	s := s*2; WRITE(B,s);
	U_2(B);

Lock-Based Concurrency Control Protocols

- **<u>Lock Types in Locking Protocols</u>**
- **Exclusive Lock** (X): The transaction holding the lock can read and write to the element, and no other transaction can acquire a lock on it.
- **Shared Lock** (S): The transaction holding the lock can only read the element, not write to it. Multiple shared locks can coexist on the same element.
- **Update Lock** (U): Initially a shared lock for reading, which can later be upgraded to an exclusive lock for writing.

Lock-Based Concurrency Control Protocols

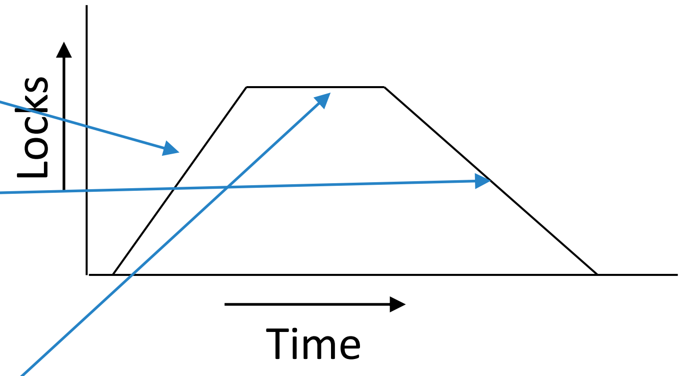
- **Lock-compatibility matrix**

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item
- But if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.

The Two Phase Locking Protocol

- A protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking Phase
  - Transaction may release locks
  - Transaction may not obtain locks



- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their lock points (i.e., the point where a transaction acquired its final lock).

Example (2PL)

T1	T2
$R_1(B)$	$R_2(B)$
B := B-50;	$R_2(A)$
$W_1(B)$	Display(A+B)
$R_1(A)$	
A := A+50;	
$W_1(A)$	

2PL

T1	T2
Lock-X(B)	Lock-S(B)
$R_1(B)$	$R_2(B)$
B := B-50;	Lock-S(A)
$W_1(B)$	$R_2(A)$
Lock-X(A)	Display(A+B)
Unlock(B)	Unlock(A)
$R_1(A)$	Unlock(B)
A := A+50;	
$W_1(A)$	
Unlock(A)	

(Growing Phase)

(Shrinking Phase)

Generate and execute the schedule: T1 starts before T2.

T1	T2
Lock-X(B);	Lock-S(B): wait
$R_1(B)$;	
B := B-50;	
$W_1(B)$	
Lock-X(A)	
Unlock(B)	
$R_1(A)$;	Lock-S(B);
A := A+50;	$R_2(B)$
$W_1(A)$	
Unlock(A)	
	Lock-S(A)
	$R_2(A)$;
	Display(A+B)
	Unlock(A);
	Unlock(B)

Example (2PL)

The two-phase locking protocol is a protocol that can potentially lead to **deadlock**!

T1	T2
Lock-X(B)	Lock-S(A)
$R_1(B)$	Lock-S(B)
B := B-50;	$R_2(B)$
$W_1(B)$	$R_2(A)$
	Display(A+B)
Lock-X(A)	Unlock(A)
Unlock(B)	Unlock(B)
$R_1(A)$	
A := A+50;	
$W_1(A)$	
Unlock(A)	

Generate and execute the schedule: T1 starts before T2.

T1	T2
Lock-X(B);	Lock-S(A)
$R_1(B)$;	
B := B-50;	
$W_1(B)$	
Lock-X(A): wait	Lock-S(B): wait
Unlock(B)	
	$R_2(B)$
$R_1(A)$;	$R_2(A)$;
A := A+50;	Display(A+B)
$W_1(A)$	Unlock(A);
Unlock(A)	Unlock(B)

Deadlock

- Neither T1 nor T2 can make progress — executing Lock-S(B) causes T2 to wait for T1 to release its lock on B, while executing Lock-X(A) causes T1 to wait for T2 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T1 or T2 must be rolled back and its locks released.

T1	T2
Lock-X(B) ;	Lock-S(A)
R_1(B) ;	
B := B-50;	
W_1(B)	
Lock-X(A): wait	Lock-S(B): wait
Unlock(B)	
R_1(A) ;	R_2(B) R_2(A) ;
A := A+50;	Display(A+B)
W_1(A)	Unlock(A) ;
Unlock(A)	Unlock(B)

Exercise

- Add lock and unlock instructions to transactions T_1 and T_2 so that they observe the two-phase locking protocol.
- Then consider: can the execution of these transactions result in a deadlock?

T_1:	T_2:
________	________
read(A)	read(B)
________	________
read(B)	read(A)
if A = 0	if B = 0
then B := B + 1	then A := A + 1
write(B)	write(A)
________	________
________	________

Exercise

- Add lock and unlock instructions to transactions T_1 and T_2 so that they observe the two-phase locking protocol.
- Then consider: can the execution of these transactions result in a deadlock?

T_1:

Lock-S(A)

read(A)

Lock-X(B)

read(B)

if A = 0

then B := B + 1

write(B)

Unlock(B)

Unlock(A)

T_2:

Lock-S(B)

read(B)

Lock-X(A)

read(A)

if B = 0

then A := A + 1

write(A)

Unlock(A)

Unlock(B)

Exercise

- Add lock and unlock instructions to transactions T_1 and T_2 so that they observe the two-phase locking protocol.
- Then consider: can the execution of these transactions result in a deadlock?

T_1:

Lock-S(A)

read(A)

Lock-X(B)

read(B)

if A = 0

then B := B + 1

write(B)

Unlock(B)

Unlock(A)

T_2:

Lock-X(A)

Lock-S(B)

read(B)

read(A)

if B = 0

then A := A + 1

write(A)

Unlock(A)

Unlock(B)