

COMP 3311

DATABASE MANAGEMENT

SYSTEMS

TUTORIAL 10

TRANSACTIONS &

CONCURRENCY CONTROL

REVIEW: TRANSACTIONS

- A **transaction** is a collection of operations that forms *a single logical unit* of work.
- The main purpose of a transaction is to **ensure that the database remains in a consistent state in the presence of multiple and concurrent operations**.
- **ACID** properties of transactions:
 - Atomicity**: either all operations succeed or none succeed.
 - Consistency**: preserve database consistency.
 - Isolation**: transactions are unaware of each other.
 - Durability**: results of operations persist.
- Two main issues to deal with:
 - **Concurrent execution** of **multiple transactions**.
 - Various kinds of **failures**, such as hardware failures and system crashes.

REVIEW: SERIALIZABILITY

Schedule: chronological execution order of transaction operations.

- Must **include all operations** of a transaction.
- Must **preserve the order of operations for each transaction**.

Serializability

Conflict equivalent schedules: transforming a schedule S into a schedule S' by a series of **swaps of non-conflicting operations**.

Conflict serializable schedule: a schedule that is **conflict equivalent to a serial schedule**.

Testing for Serializability

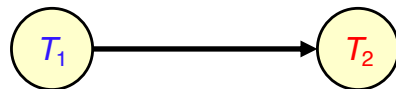
Precedence graph — a directed graph where the **vertices are the transactions** (names) and there is an **edge from T_i to T_j** if they conflict on a data item and **T_i accessed the data item earlier than T_j** .

👉 A schedule is **conflict-serializable if and only if** its **precedence graph of committed transactions does not have a cycle**.

EXAMPLE: SERIALIZABILITY

Schedule 1

T_1	T_2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	



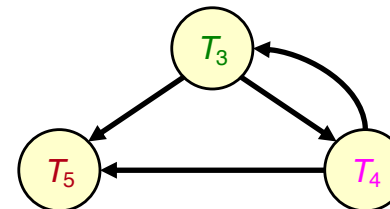
Precedence Graph

Is it conflict serializable?

Yes Equivalent to T_1, T_2 .

Schedule 2

T_3	T_4	T_5
read(Q)		
	write(Q)	
write(Q)		
		write(Q)



Precedence Graph

Is it conflict serializable?

No There is a cycle in the precedence graph.

REVIEW: RECOVERABILITY

Recoverable Schedule

- If a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i must appear before the commit operation of T_j .

Non-recoverable Schedule

- The following schedule is not recoverable if T_2 commits immediately after read(A). Why?

T_1	T_2
write(A)	read(A)
read(B)	commit
abort	

T_2 reads data item A written by T_1 and then T_1 aborts. Therefore, the data item read by T_2 is not valid.

EXAMPLE: RECOVERABILITY

Cascading Rollback

- A single transaction failure leads to a series of transaction rollbacks.
- Can lead to the undoing of a significant amount of work.

 It is highly desirable for a schedule to be cascadeless.

T_1	T_2	T_3
read(A) write(A)		
	read(A) write(A)	
		read(A)
abort		
	commit	
		commit

If T_1 fails, T_2 and T_3 must be rolled back.



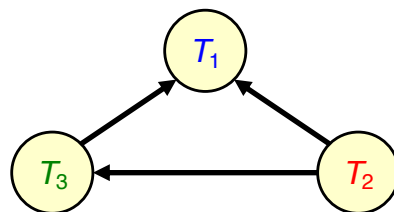
T_1	T_2	T_3
read(A) write(A) commit		
	read(A) write(A)	
	commit	
		read(A)
		commit

Cascadeless \Rightarrow If transactions commit *before* other transactions access what they have written.

EXERCISE 1

For the following schedule, state whether it is **serializable**, **recoverable** and **cascadeless**. Justify your answers.

T_1	T_2	T_3
read(X)		read(Y) read(Z)
read(Y)	read(Y)	
write(Y)	read(Z)	
write(X)		write(Z)



Precedence Graph

Serializable?

Yes The precedence graph has no cycle. The equivalent serial schedule is: T_2 , T_3 , T_1 .

Recoverable?

Yes No transaction reads a data item *previously* written by another transaction.

Cascadeless?

Yes No transaction reads a data item *previously* written by another transaction.

REVIEW: CONCURRENCY CONTROL PROTOCOLS

Locking: Shared and exclusive locks prevent multiple transactions from simultaneously accessing the same data item.

2PL (2-phase locking) idea:

Phase 1: request locks.

Phase 2: release locks.

After you unlock any data item, you cannot lock any more data items.

Timestamps: Unique ids (timestamps) assign different priorities to transactions based on the time of their submission.

TS ordering idea:

- (a) A write will fail if the timestamp of the transaction is smaller than the Read-TS or the Write-TS of the data item being written.
- (b) A read will fail if the Write-TS of the data item is larger than that of the transaction.

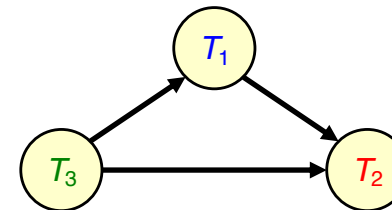
REVIEW: CONCURRENCY CONTROL PROTOCOLS

- **Multi-version protocols** use different versions of the same data.
 - A **read never fails** because a transaction can always find an appropriate version of the data item.
 - A **write fails** if the **Read-TS** of the appropriate version is **larger than** that of the transaction meaning that the value that we are trying to write has already been read by a subsequent transaction. Therefore, the write is not valid.
- 👉 **Multi-version timestamp ordering allows more schedules than simple timestamp ordering.**

EXERCISE 2

Show that the following schedule is conflict serializable and give the timestamp-ordering, serializable schedule (i.e., assign timestamps to T_1 , T_2 and T_3 so that the schedule is serializable).

T_1	T_2	T_3
read(X) write(X)		read(Y) read(Z)
		write(Y) write(Z)
read(Y) write(Y)	read(Z)	
	read(Y) write(Y) read(X) write(X)	



Precedence Graph

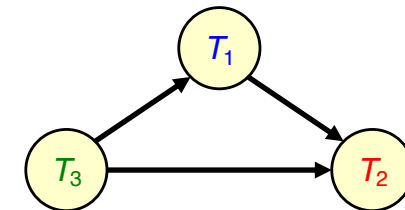
The equivalent serial schedule is: T_3, T_1, T_2 .

EXERCISE 2 (CONTD)

Timestamp (TS) ordering serializable schedule:

T_1 [TS=2], T_2 [TS=3], T_3 [TS=1]

T_1 [TS=2]	T_2 [TS=3]	T_3 [TS=1]
read(X) RTS(X)=2 write(X) WTS(X)=2 read(Y) RTS(Y)=2 write(Y) WTS(Y)=2	 read(Z) RTS(Z)=3 read(Y) RTS(Y)=3 write(Y) WTS(Y)=3 read(X) RTS(X)=3 write(X) WTS(X)=3	read(Y) RTS(Y)=1 read(Z) RTS(Z)=1 write(Y) WTS(Y)=1 write(Z) WTS(Z)=1



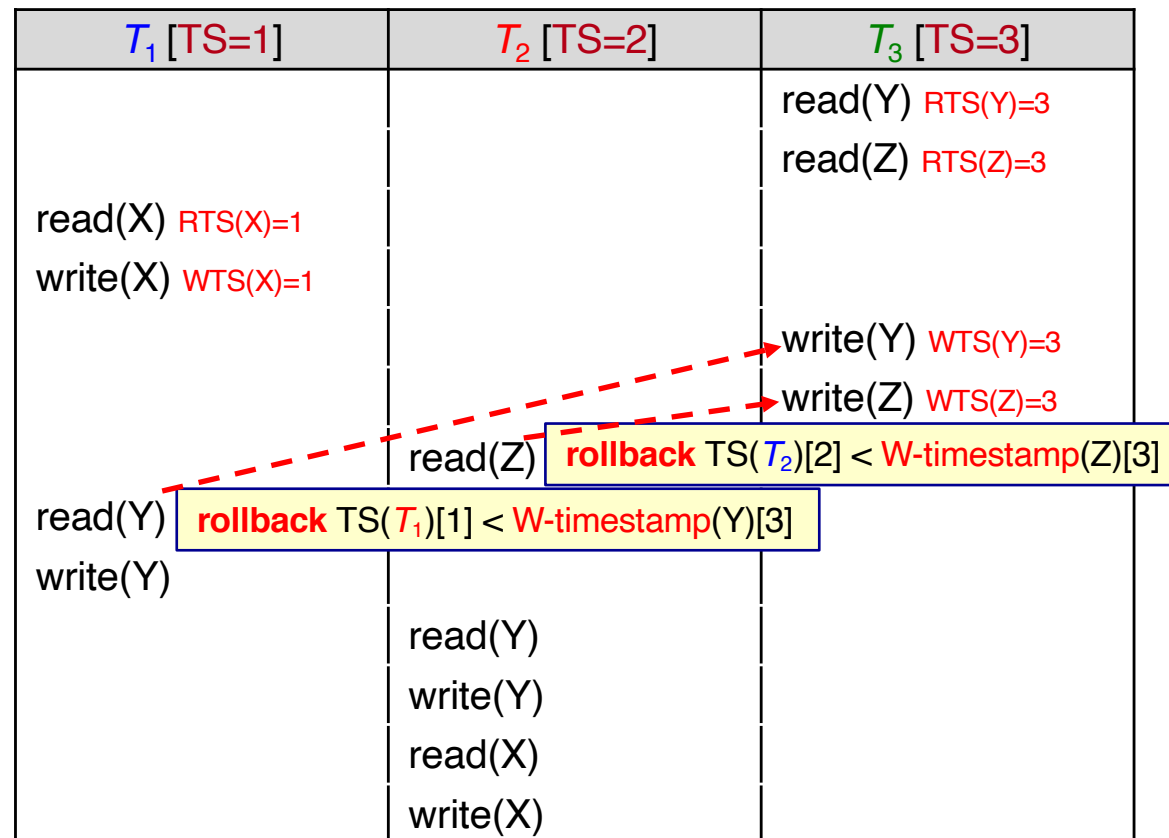
Precedence Graph

The equivalent serial schedule is: T_3 , T_1 , T_2 .

EXERCISE 2 (CONTD)

Timestamp (TS) ordering non-serializable schedule.

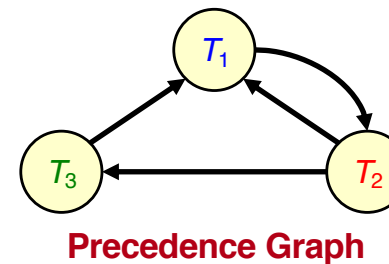
Although the schedule is serializable, only the equivalent serial order T_3 , T_1 , T_2 is allowed by the timestamp-ordering algorithm. Any other order (e.g., T_1 , T_2 , T_3) will fail as shown below.



EXERCISE 3

Is the following schedule conflict serializable? If yes, give the equivalent serial schedule. If no, show, using 2PL, how and where the schedule fails.

T_1	T_2	T_3
	read(Z) read(Y) write(Y)	
read(X) write(X)		read(Y) read(Z)
		write(Y) write(Z)
read(Y) write(Y)	read(X)	
	write(X)	

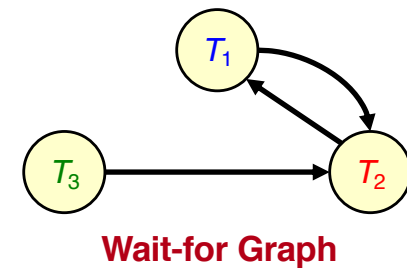


Is it conflict serializable?

No Cycle in precedence graph.

EXERCISE 3 (CONTD)

T_1	T_2	T_3
	lock-S(Z) read(Z) lock-S(Y) read(Y) lock-X(Y) write(Y)	
		lock-S(Y) → WAIT for T_2 read(Y) read(Z)
lock-S(X) read(X) lock-X(X) write(X)		
		write(Y) write(Z)
lock-S(Y) – WAIT for T_2 read(Y) write(Y)	lock-S(X) → WAIT for T_1 read(X)	
	write(X)	



T_3 waiting for T_2
 T_2 waiting for T_1
 T_1 waiting for T_2

Deadlock!

EXERCISE 4

Consider the following schedule consisting of transactions T_1 , T_2 , T_3 and T_4 (note: r_1 means T_1 read, w_1 means T_1 write and so on):

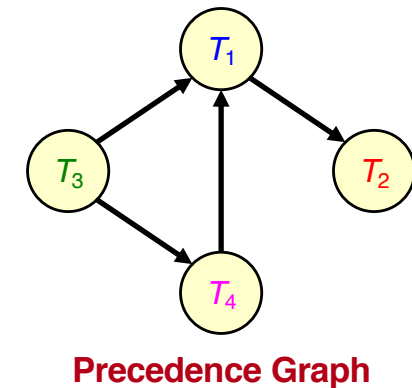
Schedule: $r_1(X)$, $w_1(X)$, $r_2(X)$, $r_3(Y)$, $w_3(Y)$, $w_2(X)$, $r_4(Y)$, $w_1(Y)$

- a) Show that the schedule is conflict serializable by constructing the precedence graph.
- b) What is the equivalent serial schedule?
- c) Can the schedule be rewritten so it becomes recoverable, but not cascadeless by adding commit operations in the appropriate locations in the schedule? Explain.
- d) Can the schedule be rewritten so it becomes both recoverable, and cascadeless by adding commit operations in the appropriate locations in the schedule? Explain.

EXERCISE 4 (CONTD)

- a) Show that the schedule is conflict serializable by constructing the precedence graph.

T_1	T_2	T_3	T_4
read(X) write(X)	read(X) write(X)	read(Y) write(Y)	 read(Y)



- b) What is the equivalent serial schedule? T_3, T_4, T_1, T_2

EXERCISE 4 (CONTD)

- c) Can the schedule be rewritten so it becomes recoverable, but not cascadeless by adding commit operations in the appropriate locations in the schedule? Explain.

Recall: A schedule is recoverable if the **commit** of a transaction T_j that reads data items *previously written* by a transaction T_i appears after the commit operation of T_i .

T_2 reads X written by T_1
 $\Rightarrow T_2$ must commit after T_1 .

T_4 reads Y written by T_3
 $\Rightarrow T_4$ must commit after T_3 .

Schedule:

$r_1(X), w_1(X), r_2(X), r_3(Y), w_3(Y),$
 $w_2(X), r_4(Y), w_1(Y), c_1, c_2, c_3, c_4$

or

$r_1(X), \dots, c_3, c_4, c_1, c_2$

or

$r_1(X), \dots, c_3, c_1, c_4, c_2$

T_1	T_2	T_3	T_4
read(X)			
write(X)			
	read(X)		
		read(Y)	
		write(Y)	
	write(X)		
			read(Y)
write(Y)			

EXERCISE 4 (CONTD)

- d) Can the schedule be rewritten so it becomes both recoverable, and cascadeless by adding commit operations in the appropriate locations in the schedule? Explain.

Recall: A schedule is cascadeless if, for each pair of transactions T_i , T_j such that T_j reads a data item previously written by T_i , the **commit** operation of T_i appears before the **read** operation of T_j .

- The commit of T_1 must appear before the read(X) of T_2 .
- The commit of T_3 must appear before the read(Y) of T_4 .

The schedule cannot be made cascadeless as written.

To make the schedule cascadeless, the write(Y) of T_1 has to be moved after the write(X) of T_1 so that T_1 can commit before the read(X) of T_2 .

T_1	T_2	T_3	T_4
read(X)			
write(X)	read(X)		
		read(Y)	
	write(X)	write(Y)	
			read(Y)
write(Y)			