COMP 3311
DATABASE MANAGEMENT SYSTEMS

LECTURE 21
CONCURRENCY CONTROL:
TIMESTAMP-BASED PROTOCOLS

TIMESTAMPS

Each transaction is issued a fixed timestamp TS(T_i) by the system before it starts execution.

If an old transaction T_i has timestamp TS(T_i), a new transaction T_j is assigned timestamp TS(T_j) such that TS(T_i) < TS(T_j).

☞ **The timestamps determine the serializability order.**

- To implement serializability, two timestamp values are associated with each data item Q.

WTS(Q) (write timestamp) the largest timestamp of any transaction that executed write(Q) successfully.

RTS(Q) (read timestamp) the largest timestamp of any transaction that executed read(Q) successfully.

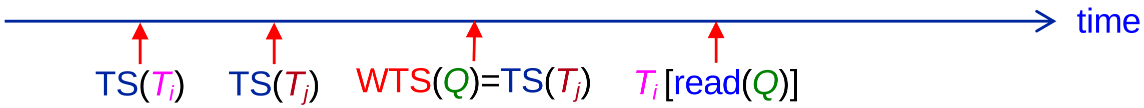
The timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

TIMESTAMP-ORDERING PROTOCOL: READ

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.

Suppose a transaction T_i issues a read(Q).

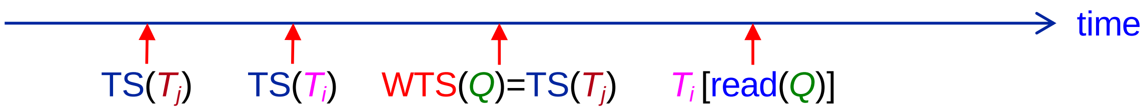
1. If TS(T_i) < WTS(Q), then T_i needs to read a value of Q that was already overwritten by a newer transaction T_j that started <u>after</u> T_i.



☞ Reject the read operation and **roll back** T_i.

☞ Restart T_i with a new (larger) timestamp TS′(T_i).

2. If TS(T_i) ≥ WTS(Q), then execute the read operation and set RTS(Q) to the *maximum* of RTS(Q) and TS(T_i) since Q was written by an older transaction T_j that started <u>before</u> T_i.

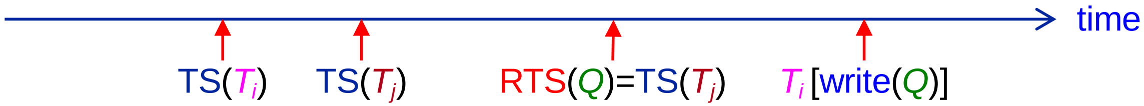


☞ **T_i can read a data item Q iff TS(T_i) ≥ WTS(Q).**

TIMESTAMP-ORDERING PROTOCOL: WRITE

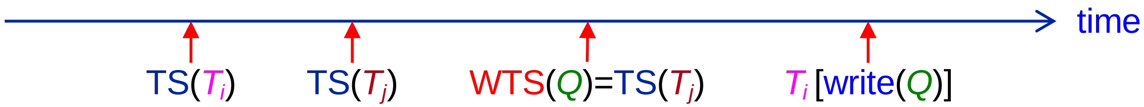
Suppose that transaction T_i issues a write(Q).

1. If TS(T_i) < RTS(Q), then the value of Q that T_i is writing was needed previously and the system assumed it would never be written (since a newer transaction T_j that started after T_i already read a different value).



☞ Reject the write operation and **roll back** T_i.

2. If TS(T_i) < WTS(Q), then T_i is attempting to write an obsolete value of Q since a newer transaction T_j that started after T_i wrote a newer value of Q.



☞ Reject the write operation and **roll back** T_i.

3. Otherwise, execute the write operation and set WTS(Q) to TS(T_i).

☞ **T_i can write a data item Q iff TS(T_i) > RTS(Q) and WTS(Q).**

TIMESTAMP-ORDERING PROTOCOL EXAMPLE

Below is a partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5. Assume initial R/W timestamp of all items is 0.

RTS(X)=5	WTS(X)=0	RTS(Y)=2	WTS(Y)=3	RTS(Z)=5	WTS(Z)=3

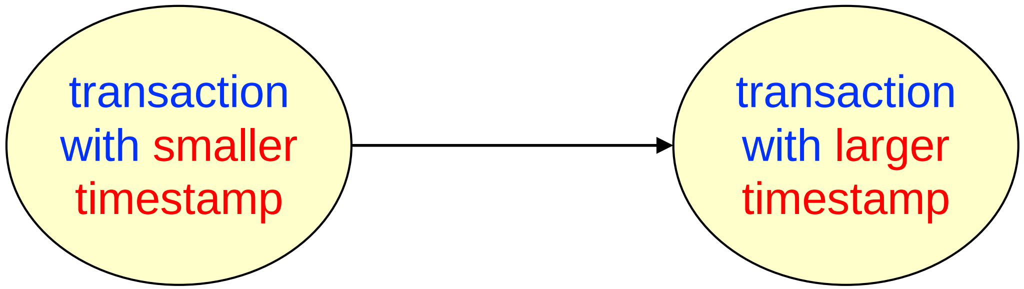
T_1 [TS=1]	T_2 [TS=2]	T_3 [TS=3]	T_4 [TS=4]	T_5 [TS=5]
				read(X) ✓ $TS(T_5)=5 \geq WTS(X)=0$; set $RTS(X)=5$
	read(Y) ✓ $TS(T_2)=2 \geq WTS(Y)=0$; set $RTS(Y)=2$			
read(Y) ✓ $TS(T_1)=1 \geq WTS(Y)=0$				
		write(Y) ✓ $TS(T_3)=3 \geq RTS(Y)=2$ & $\geq WTS(Y)=0$; set $WTS(Y)=3$		
		write(Z) ✓ $TS(T_3)=3 \geq RTS(Z)=0$ & $WTS(Z)=0$; set $WTS(Z)=3$		
	read(Z) $TS(T_2)=2 < WTS(Z)=3$ ⟹ rollback			
				read(Z) ✓ $TS(T_5)=5 \geq WTS(Z)=3$; set $RTS(Z)=5$
				⟹ rollback
read(X) ✓ $TS(T_1)=1 \geq WTS(X)=0$				
		write(Z) $TS(T_3)=3 < RTS(Z)=5$ ⟹ rollback		
				write(Y)
				write(Z)

Read
If $TS(T_i) < WTS(Q)$ **rollback**
If $TS(T_i) \geq WTS(Q)$
$RTS(Q) = \max(TS(T_i), RTS(Q))$

Write
If $TS(T_i) < RTS(Q)$ **rollback**
If $TS(T_i) < WTS(Q)$ **rollback**
Otherwise $WTS(Q) = TS(T_i)$

TIMESTAMP-ORDERING PROTOCOL CORRECTNESS

- The timestamp-ordering protocol ensures conflict serializability since all the arcs in the precedence graph are of the form:



☞ **Conflicting instructions are processed in timestamp order.**

- Since there can be no cycles in the precedence graph, the timestamp protocol ensures freedom from deadlock and no transaction ever waits.

- However, starvation of long transactions is possible, and a schedule may not be recoverable.

RECOVERABILITY AND CASCADING ROLLBACK

- The problem with the timestamp-ordering protocol:
  - Suppose T_i aborts, but T_j has read a data item written by T_i.
  - Then T_j must abort; if T_j had been allowed to commit before T_i aborted, the schedule is not recoverable.
  - Moreover, any transaction that has read a data item written by T_j must abort.

☞ **This can lead to cascading rollback.**

Solution

- A transaction is structured such that *all its writes* are performed at the end of its processing.
- All writes of a transaction form an atomic action; no transaction may execute while a transaction is being written.
- A transaction that aborts is restarted with a new timestamp.

THOMAS' WRITE RULE

Read
If TS(T_i) < WTS(Q) **rollback**
If TS(T_i) ≥ WTS(Q)
RTS(Q) = max(TS(T_i), RTS(Q))

Write
If TS(T_i) < RTS(Q) **rollback**
If TS(T_i) < WTS(Q) **rollback**
Otherwise WTS(Q) = TS(T_i)

RTS(Q)=1	WTS(Q)=2

T_1 [TS=1]	T_2 [TS=2]
read(Q) ✓ TS(T_1)=1 ≥ WTS(Q)=0; set RTS(Q)=1	
	write(Q) ✓ TS(T_2)=2 ≥ RTS(Q)=1 & ≥ WTS(Q)=0; set WTS(Q)=2
write(Q) TS(T_1)=1 < WTS(Q)=2 ⟹ rollback	

☞ **Rollback of T_1 is unnecessary!**

- Since T_2 has already written Q (with TS=2), the value T_1 is attempting to write (with TS=1) *will never be read*.
  - Any transaction with timestamp less than TS(T_2) that attempts to read Q will be rolled back since its timestamp will be less than WTS(Q).
  - Any transaction with a timestamp greater than TS(T_2) must read the value of Q written by T_2, rather than the one T_1 is attempting to write.

☞ **Ignore write of T_1!**

Read (unchanged)
If TS(T_i) < WTS(Q) **rollback**
If TS(T_i) ≥ WTS(Q)
RTS(Q) = max(TS(T_i), RTS(Q))

Write (Thomas' write rule)
If TS(T_i) < RTS(Q) **rollback**
If TS(T_i) < WTS(Q) **ignore**
Otherwise WTS(Q) = TS(T_i)

VALIDATION-BASED PROTOCOLS

Assumption: Most transactions are read-only.

☞ **Conflicts among transactions are rare.**

- Transactions are monitored by executing in two or three phases:
  1. **Read phase** write data items in temporary, local variables.
  2. **Validation phase** determine whether the transaction can proceed to the write phase; abort transaction if test fails.
  3. **Write phase** apply write operations to database. Read-only transactions omit this phase.

- The system maintains three timestamps for each transaction:
  1. start(T_i) the time when T_i started execution.
  2. validation(T_i) the time when T_i finished its read phase and started its validation phase.
  3. finish(T_i) the time when T_i finished its write phase.

  ☞ **Serializability is according to validation time-stamp ordering.**

VALIDATION TEST

- For all transactions T_k with TS(T_k) < TS(T_i), one of the following conditions must hold to pass the validation test.

  1. finish(T_k) < start(T_i).

     Serializable since T_k completes before T_i started.

  2. data items written by (T_k) ∩ data items read by (T_i) = ∅ (no overlap) and (start(T_i) < finish(T_k) < validation(T_i) (T_k does its writes before validation phase of $T_i \Longrightarrow$ writes of T_k and T_i do not overlap).

     Serializable since the writes of T_k do not affect the read of T_i, and T_i cannot affect the read of T_k.

- Transactions that fail the test are rolled back and restarted.

- Ensures no cascading rollbacks, but starvation of long transactions is possible $\Longrightarrow$ may need to block short transactions to allow a long transaction to complete.

☞ **Known as optimistic concurrency-control.**

VALIDATION-BASED PROTOCOL EXAMPLE

Suppose that $TS(T_1) < TS(T_2)$

– data items written by (T_1)

$\cap$

data items read by (T_2) = Ø (no overlap)

– (start(T_2) < finish(T_1) < validation(T_2)

(T_1 does no writes)

- Note that the writes to the actual variables are performed only after the validation phase of T_2.
- Thus, T_1 reads the old values of A and B, and this schedule is serializable.

☞ Serial schedule: T_1 T_2

T_1	T_2
read(B)	
	read(B)
	B := B - 50
	read(A)
	A := A - 50
read(A)	
validate	
display(A+B)	
	validate
	write(B)
	write(A)

MULTIVERSION SCHEMES

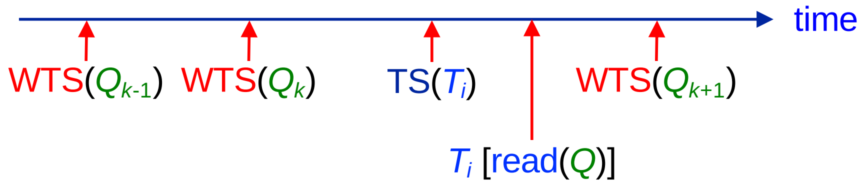
- Multiversion schemes keep old versions of data items, labelled with timestamps, to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each data item Q has a sequence of versions $<Q_1, Q_2, \ldots, Q_m>$.
- Each version Q_k contains three data fields:

Content	The value of version Q_k.
WTS(Q_k)	The timestamp of the transaction that created (wrote) version Q_k.
RTS(Q_k)	The largest timestamp of a transaction that successfully read version Q_k.

- Each successful write by a transaction T_i creates a new version Q_k of Q and sets Q_k's RTS and WTS to TS(T_i).

MULTIVERSION TIMESTAMP ORDERING: READ

Let Q_k be the version of Q whose write timestamp is the largest write timestamp *less than or equal to* TS(T_i).



- If transaction T_i issues a read(Q), then the value returned is the content of version Q_k (i.e., T_i reads the most recent version that comes before it in time).

  ☞ If TS(T_i) > RTS(Q_k), then set RTS(Q_k) = TS(T_i).

☞ **Reads never fail and never wait as an appropriate version can always be found.**

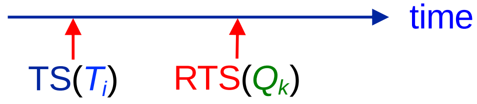
Read
Reads always succeed
set RTS(Q_k) = max(TS(T_i), RTS(Q_k))

MULTIVERSION TIMESTAMP ORDERING: WRITE

Let Q_k be the version of Q whose write timestamp is the largest write timestamp *less than or equal to* TS(T_i).

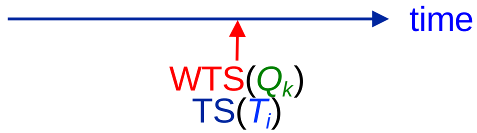
If transaction T_i issues a write(Q), then

a) if TS(T_i) < RTS(Q_k), then **roll back** T_i.

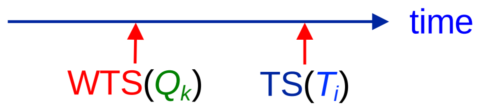


Since some other transaction T_j that should read T_i's write (in the serialization order defined by the timestamp values) has already read a version created by a transaction older than T_i.

b) if TS(T_i) = WTS(Q_k), overwrite the contents of Q_k $\Longrightarrow$ Q_k was written before also by T_i.



c) if TS(T_i) > WTS(Q_k) create a new version of Q and set R/WTS of new version to TS(T_i).



☞ **Conflicts are resolved through aborting transactions.**

Write
If TS(T_i) < RTS(Q_k) **rollback**
If TS(T_i) = WTS(Q_k) **overwrite contents**
If TS(T_i) > WTS(Q_k) **create new version**
set R/WTS(Q')=TS(T_i)

MULTIVERSION SCHEMES: NOTES

- The schedule of **Exercise 2** will terminate successfully although it is not conflict serializable.
- The concept of conflict serializability <u>does not apply</u> to multi-version protocols, since read operations do not conflict with previous writes (there is always a version to read).

Versions are removed as follows:

- Suppose that there are two versions Z_1 and Z_2 with timestamps 3 and 1 (as could happen in **Exercise 2**) that both have a smaller timestamp than the oldest transactions (i.e., T_1, T_2 and T_3 have committed and the new transactions have timestamps ≥ 4).
- Then the version Z_2 with timestamp 1 will not be used again and is deleted.
- If any subsequent transaction needs to read Z, it will read Z_1 with timestamp 3.

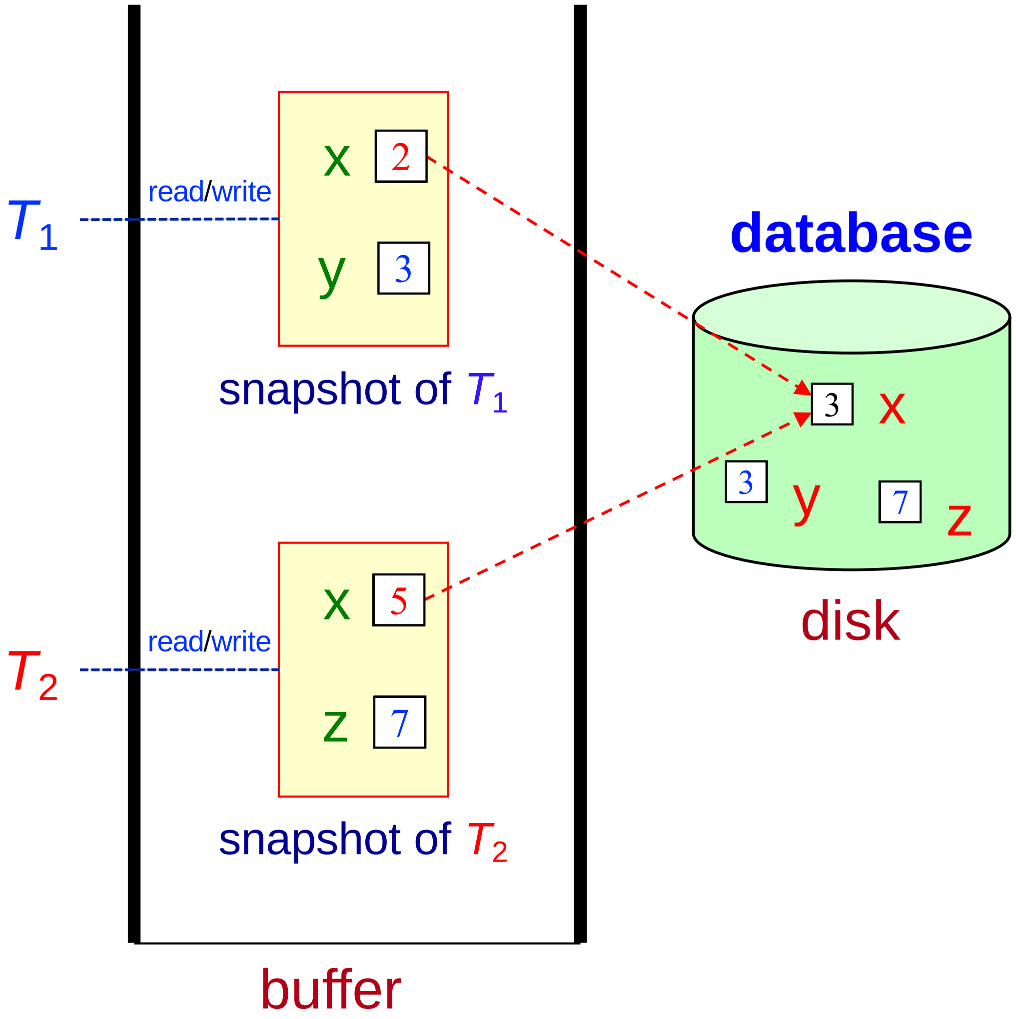
- In **Exercise 2** it is as if the last write(Z) was never executed, which is OK because T_3 executes a *blind* write(Z) (i.e., write with no previous read).
- If, on the other hand, T_3 had read Z *<u>before</u>* writing it, the schedule would fail at the last write of T_1.

SNAPSHOT ISOLATION

- Each transaction works on its own private copy (snapshot) of the data items it reads and writes.
- If most transactions are read-only or if their updated data items do not overlap, then they cannot conflict with each other.

  ☞ **Results in higher concurrency levels.**

- Concurrency control is only needed if updates conflict.



SNAPSHOT ISOLATION VALIDATION STEPS

First Committer Wins

if a data item that *T* intends to write has already been updated by any concurrent transaction:
then *T* **aborts**;
else *T* **commits** and writes its updates to the database;

First Updater Wins

if *T*'s request for an x-lock on a data item it intends to update is granted:
    if the data item has already been updated by any concurrent transaction:
        then *T* **aborts**;
        else *T* **proceeds with its execution** including possibly committing;
else // data item is locked ⟹ wait until transaction holding the lock aborts or commits
    if the transaction holding the lock aborts:
        then the lock is released, and *T* locks the data item;
        if the data item has already been updated by any concurrent transaction:
            then *T* **aborts**;
            else *T* **proceeds with its execution** including possibly committing;
    if the transaction holding the lock commits:
        then *T* **aborts**;

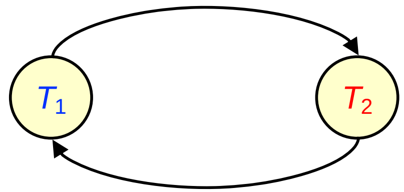
SNAPSHOT ISOLATION: NOTES

- Snapshot isolation is attractive because overhead is low, and no abort occurs unless two concurrent transactions update the same data item.

- Snapshot isolation does not ensure serializability!
  - The schedule is allowed by snapshot isolation since T_1 and T_2 update different data items.
  - However, it is not serializable!

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

Precedence Graph



- Although snapshot isolation permits schedules that are not serializable, it is widely used in commercial systems (e.g., Oracle, SQL Server).
  - If database consistency is preserved, then non-serializable executions are OK.
  - Constraints can be used to help ensure database consistency when non-serializable schedules are allowed.

CONCURRENCY CONTROL: SUMMARY

While most protocols discussed ensure correctness,

☞ **A correct schedule may not be permitted by a protocol.**

☞ **The more correct schedules allowed by a protocol, the greater the degree of concurrency.**

- The protocols differ in the way they handle conflicts:

  i. Lock-based protocols make transactions wait (thus, they can result in deadlocks).

  ii. Timestamp-based protocols make transactions abort (thus, there are no deadlocks, but aborting a transaction may be more expensive).

- Recoverability (i.e., no rollback after a commit) is a necessary property of a schedule.

  ☞ **To ensure recoverability, transaction T_i can commit only after all transactions that wrote items that T_i read have committed.**

CONCURRENCY CONTROL: SUMMARY (cont'd)

- A cascading rollback happens when an *uncommitted* transaction must be rolled back because it read an item written by a transaction that aborted.

☞ **It is desirable to have cascadeless schedules.**

☞ **To achieve this property a transaction should only be allowed to read items written by committed operations.**

- If a schedule is cascadeless, it is also recoverable.
  - Strict 2PL ensures cascadeless schedules by releasing all exclusive locks of transaction T_i after T_i commits (therefore other transactions cannot read the items locked by T_i at the same time).
  - Timestamp-based protocols can also achieve cascadeless schedules by performing all the writes at the end of the transaction as an atomic operation.