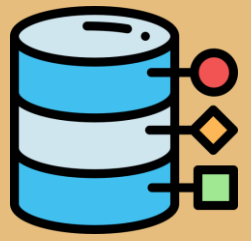


CSAI 302

Advanced Database Systems

Lec 08

Concurrency Control

Introduction

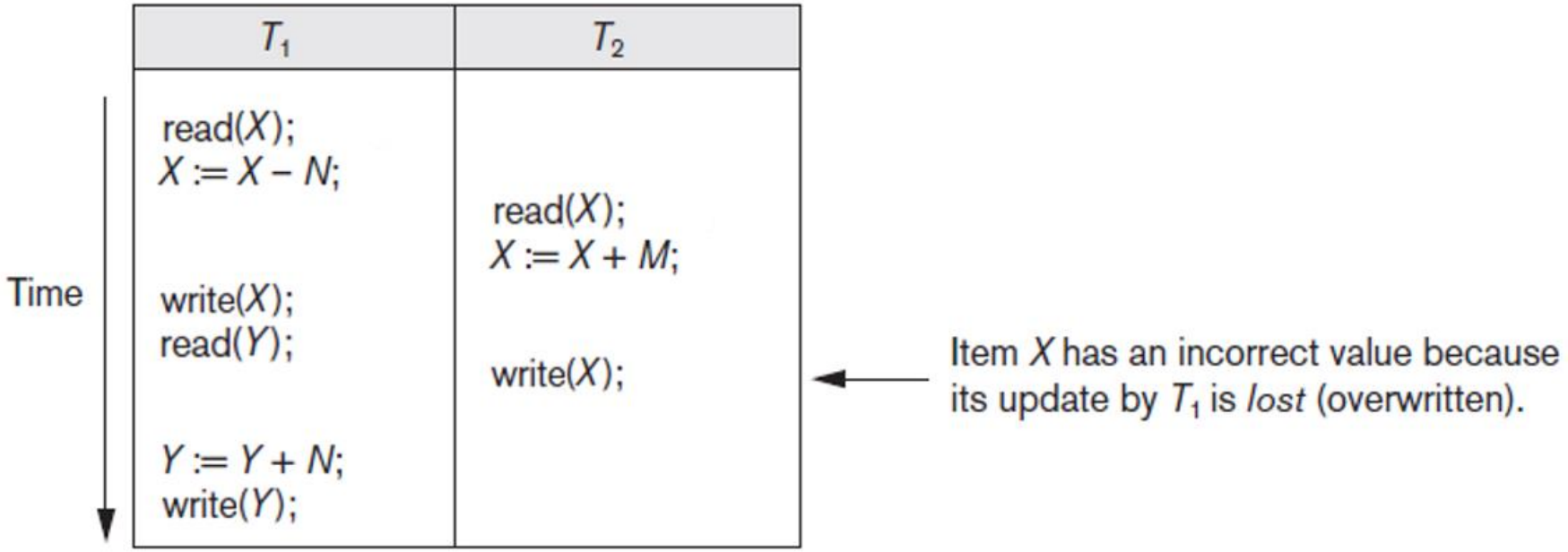
- A database must provide a mechanism that will ensure that all possible schedules are:
  - either **conflict** or **view serializable**, and
  - are **recoverable** and preferably **cascadeless**
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability after it has executed is a little too late!

Concurrency control problems

- Dirty write (or lost update)
- Dirty read (or temporary update)
- Non-repeatable read
- Incorrect summary (or phantom phenomena)

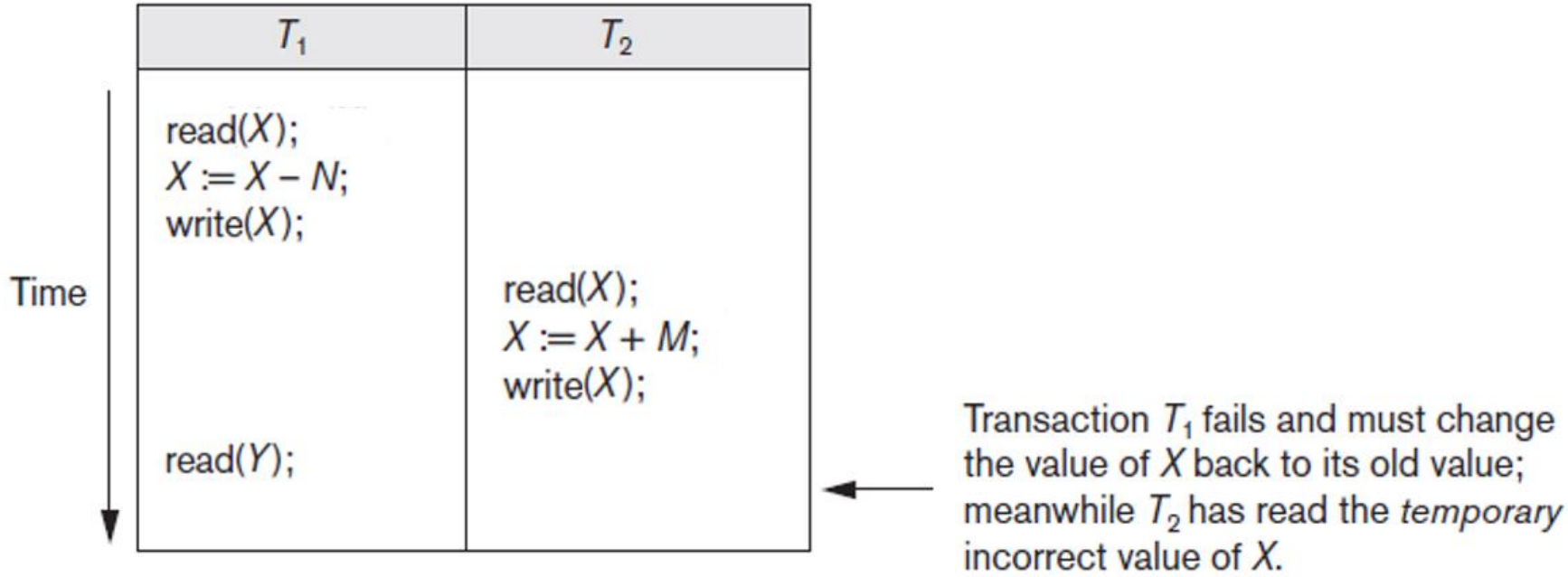
Dirty write (lost update)

- Two transactions update (read and write) the same item, second update starts before the first is complete
  - ★ (updates are interleaved)
- **Result**: incorrect value



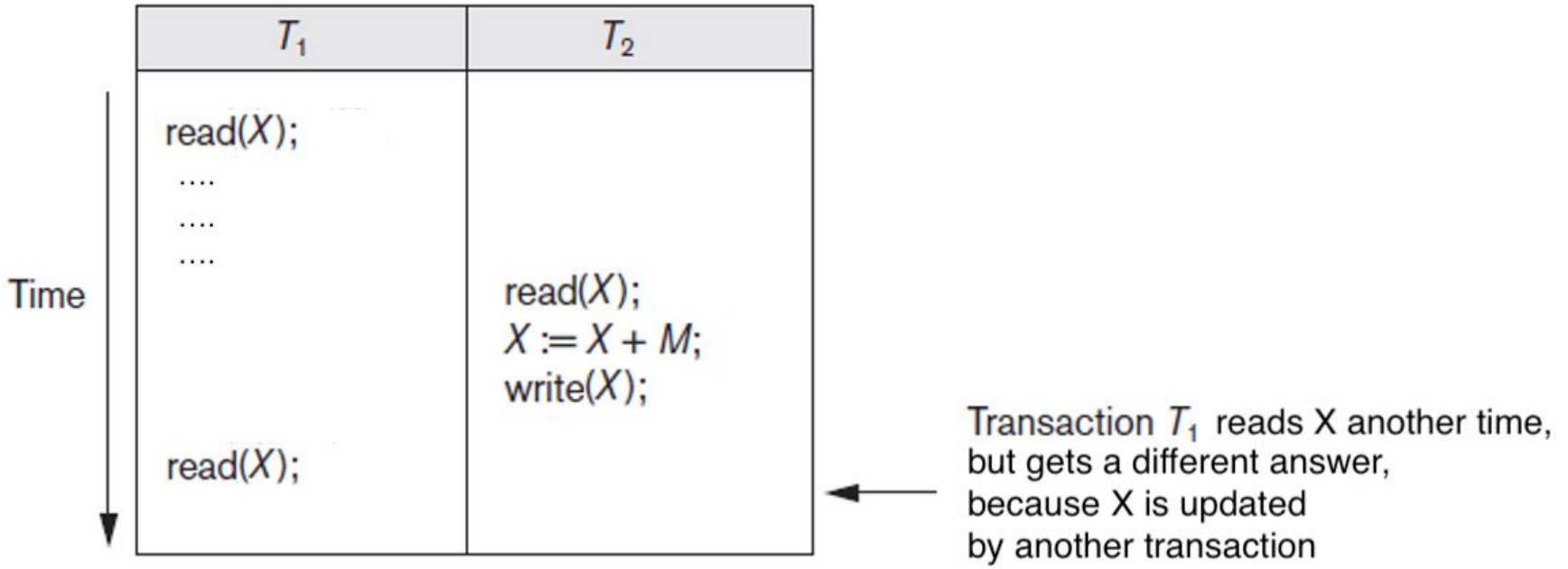
Dirty read (temporary update)

- A transaction **updates** an item,
- new value is **used** by another transaction,
- the first transaction **fails** and its update is **rolled** back
- **Result**: the second transaction relies on an incorrect value



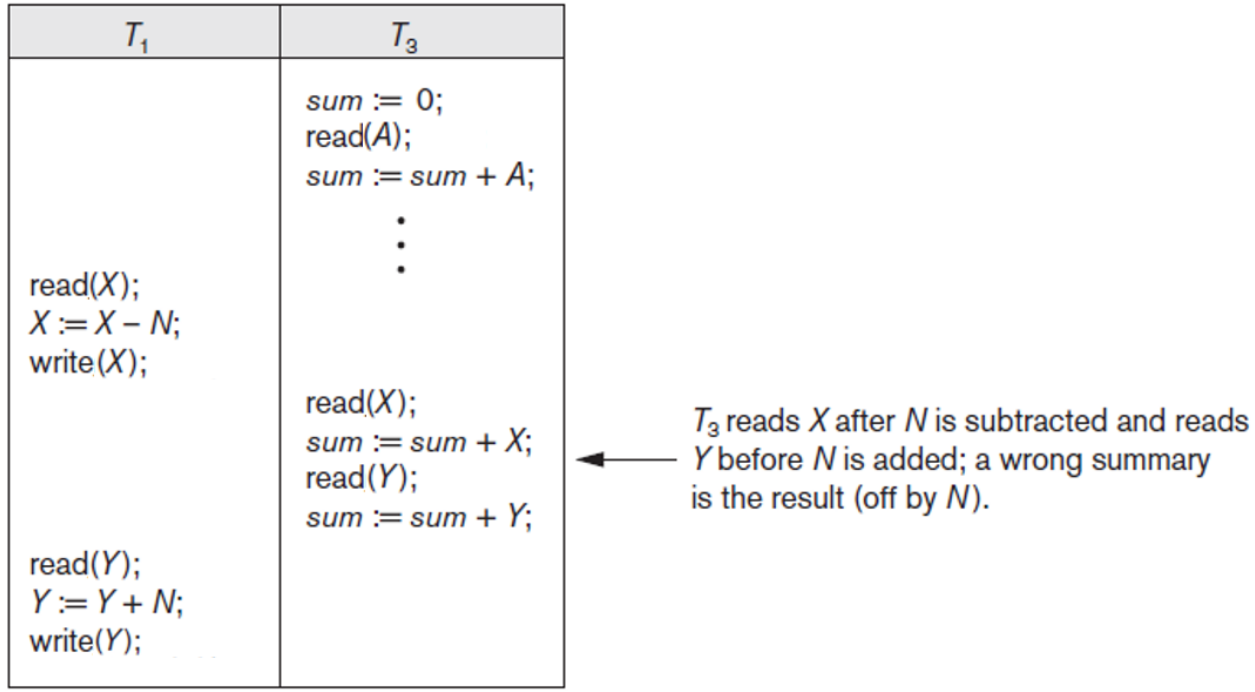
Non-repeatable read

- One transaction reads the same item **twice**
- another transaction **changes** its value in between
- **Result**: the first transaction gets different values of the same item



Incorrect summary (Phantom phenomena)

- One transaction calculates an **aggregate** summary
- while other transactions **update** some involved items
- **Result**: the aggregate is inconsistent



Why recovery is needed?

- (if something goes **wrong**, effects **roll back**)
- Should be either
  - **Committed**:
    - evaluated in full and all effects permanently recorded
  - **Aborted**:
    - no effect on the database (everything done rolled back)

The System Log

- System log keeps track of transaction operations
  - Sequential, append-only file
  - Not affected by failure
- Log buffer
  - Main memory buffer
  - When full, appended to end of log file on disk
- Log file is backed up periodically
- Undo and redo operations based on log possible

Commit Point of a Transaction

- Occurs when all operations that access the database have **completed successfully**
- Transaction writes a commit record into the log
  - If system failure occurs, can search for transactions with recorded **start_transaction** but **no commit** record

Concurrency Control

Concurrency Control (Cont.)

- Goal
  - **Develop concurrency control protocols that will assure serializability.**
- Concurrency-control schemes **tradeoff** between the ***amount of concurrency*** they allow and the ***amount of overhead*** that they incur.
  - Some schemes allow only **conflict-serializable** schedules to be generated, while others allow **view-serializable**.

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols
  - allow concurrent schedules,
  - ensure that the schedules are conflict/view serializable,
  - and are recoverable and cascadeless.
- Concurrency control protocols ***do not examine*** the precedence graph as it is being created
  - Instead, a protocol imposes a discipline that avoids non-serializable schedules.
    - **Tests for serializability help us understand why a concurrency control protocol is correct.**

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate.
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Isolation levels

- It is not always necessary to ensure no problems of all types
  - in some applications we can tolerate some problems
  - for the sake of more concurrency (i.e., efficiency)
- Simple isolation level hierarchy of schedules (and transaction management protocols):
  - **Level 0**: no dirty reads
  - **Level 1**: no dirty writes
  - **Level 2**: no dirty reads, no dirty writes
  - **Level 3**: (true isolation): no dirty reads, no dirty writes, no non-repeatable reads
- *A protocol that ensure recoverable and serializable schedules is Level 3*

Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database

Locking

Locks

- **Lock**: a variable associated with a data item
  - describes status for operations that can be applied
  - unique **Lock(X)** for each data item **X**
- binary locks:
  - **Lock(X)** takes values **locked** or **unlocked** (or 1 and 0)
- shared/exclusive locks:
  - **Lock(X)** takes values **read-locked**, **write-locked**, or **unlocked**
- These variables are managed by concurrency control subsystem

Binary locks

- for each item X, variable Lock(X) can have on of the two values:
  - ★ Lock(X) is locked (or 1) means that X is locked by a transaction and cannot be accessed by any other transaction
  - ★ Lock(X) is unlocked (or 0) means that X is available and can be accessed when requested
- schedules start with all unlocked

Binary locks in transactions

- Use two operations **lock(X)** and **unlock(X)** applying these rules:
  - all **read**(X) and **write**(X) must be between the two
  - *well-formed*: no unlocking without locking, etc.
- These operations are implemented such that
  - when **lock(X)** is issued
    - if **Lock(X) = 1**, the transaction is forced to wait for **Lock(X) = 0** (in the schedule)
    - if **Lock(X) = 0**, it is set to 1 and the transaction can proceed
  - when **unlock(X)** is issued, **Lock(X)** is set to 0

Ex1: Transactions with binary locks

T_1
lock(X);
read(X);
$X := X - 1$;
write(X);
unlock(X);
lock(Y);
read(Y);
$Y := Y + 1$;
write(Y);
unlock(Y);

T_2
lock(X);
lock(Y);
read(X);
unlock(X);
$X := X + 1$;
lock(X);
write(X);
unlock(Y);
unlock(X);

$T1$: l1(X), r1(X), w1(X), u1(X), l1(Y), r1(Y), w1(Y), u1(Y): Ok!

$T2$: l2(X), l2(Y), r2(X), u2(X), l2(X), w2(X), u2(Y), u2(Y): Ok!

Note: $T2$ locks and unlocks X twice and locks Y without a need

Ex2: Transactions with binary locks

T_1
lock(X);
read(X
$X := X - 1$;
write(X);
unlock(X);
read(Y);
lock(Y);
$Y := Y + 1$;
write(Y);
unlock(Y);

T_2
lock(X);
lock(Y);
read(X);
unlock(X);
$X := X + 1$;
lock(X);
write(X);
unlock(X);

$T1$: l1(X), r1(X), w1(X), u1(X), r1(Y), l1(Y), w1(Y), u1(Y): Not ok

$T2$: l2(X), l2(Y), r2(X), u2(X), l2(X), w2(X), u2(Y): Not ok

Note: $T1$ reads Y without locking, and $T2$ does not unlock Y

Ex: Schedules with binary locks

- $S1$: l1(X), r1(X), u1(X), l2(X), r2(X), u2(X), l1(X), w1(X), u1(X)
  - Ok!
- $S2$: l1(X), r1(X), l2(X), u1(X), r2(X), u2(X), l1(X), w1(X), u1(X)
  - Ok!
- $S3$: l1(X), r1(X), l2(X), r2(X), u1(X), u2(X), l1(X), w1(X), u1(X)
  - Not ok
    - (implementation of l2(X) is 'broken': should not allow to continue $T2$ with r2(X) before u1(X))
- Note: locks (i.e., the rules and implementations as above) do not guarantee serializability by themselves;

Lock table

- Concurrency control subsystem has a lock manager module that relies on a lock table:
  - has schema [Data_item_name, Locking_transaction]
  - keeps only locked items (others are unlocked)
- Lock manager keeps track of and controls access to locks by checking the rules for transactions and enforcing allowed schedules (waiting, etc.)
- Binary locking is too restrictive anyway

Shared/exclusive locks

- Allow sharing for several reading transactions (but exclusive writes)
- for each item X, **Lock(X)** is
  - ***read-locked*** (or ***share-locked***) means that X is read by a transaction
    - variable **#READS(X)** keeps the number of reading transactions
  - ***write-locked*** (or exclusive-locked) means that X is exclusively locked for writing by some transaction and cannot be accessed by others
  - ***unlocked*** means that X is available for access (with locking)
- schedules start with all ***Lock(X) = unlocked*** and **#READS(X)** = 0

Transaction with shared/exclusive locks

- use three operations, read_lock(X), write_lock(X) and unlock(X) with rules:
  - each read(X) is after read_lock(X) or write_lock(X) (with no unlock(X) in between)
  - each write(X) is after write_lock(X) (with no unlock(X) in between)
  - each read_lock(X) and write_lock(X) has unlock(X) at some point after
  - no changing of lock type before unlocking;
    - that is, no read_lock(X) after write_lock(X) without unlock(X) between

Implementation of read_lock

```
read_lock(X):
B:   if   LKD(X) = "unlocked"
          then begin   LKD(X) ← "read-locked";
                       #READS(X) ← 1
               end
     else if   LKD(X) = "read-locked"
          then    #READS(X) ←    #READS(X) + 1
     else begin
               wait (until   LKD(X) = "unlocked"
                        and the lock manager wakes up the transaction);
               go to B
               end;
```

Implementation of write_lock

```
write_lock(X):
B:   if   LKD(X) = "unlocked"
          then   LKD(X) ← "write-locked"
     else begin
               wait (until   LKD(X) = "unlocked"
                     and the lock manager wakes up the transaction);
               go to B
               end;
```

Implementation of unlock

```
unlock (X):
    if   LKD(X) = "write-locked"
         then begin   LKD(X) ← "unlocked";
                      wakeup one of the waiting transactions, if any
                      end
    else it   LKD(X) = "read-locked"
         then begin
                      #READS(X) ←    #READS(X) –1;
                      if   #READS (X) = 0
                           then begin   LKD(X) = "unlocked";
                                        wakeup one of the waiting transactions, if any
                                        end
                      end;
```

Examples

- $S1$: rl1(X), r1(X), u1(X), rl2(X), r2(X), u2(X), wl1(X), w1(X), u1(X)
  - Ok!
- $S1'$: rl1(X), r1(X), u1(X), rl2(X), r2(X), u2(X), rl1(X), w1(X), u1(X)
  - Not ok (write with read lock)
- $S1''$: wl1(X), r1(X), u1(X), rl2(X), r2(X), u2(X), wl1(X), w1(X), u1(X)
  - Ok!
- $S3$: rl1(X), r1(X), rl2(X), r2(X), u1(X), u2(X), wl1(X), w1(X), u1(X)
  - Ok!
- $S3'$: rl1(X), r1(X), wl2(X), w2(X), u1(X), u2(X), wl1(X), w1(X), u1(X)
  - Not ok (implementation of wl2(X) is 'broken': should be exclusive)
- $S3''$: rl1(X), r1(X), rl2(X), r2(X), u2(X), wl1(X), w1(X), u1(X)
  - Not ok ($T1$ is not following the rules: a second lock without unlocking)

Two-phase locking

Two-phase locking

- A transaction (binary or shared/exclusive) follows **two-phase locking protocol** if:
  - all **locking** operations are ***before*** all **unlocking** operations
- Two phases:
  - locking (**first**, **expanding**, or **growing**) phase
  - unlocking (**second** or **shrinking**) phase

Two-phase locking: negative example

T_1	T_2

read_lock(Y);
read(Y);
unlock(Y);
write_lock(X);
read(X);
$X := X + Y$;
write(X);
unlock(X);
	read_lock(X);
	read(X);
	unlock(X);
	write_lock(Y);
	read(Y);
	$Y := X + Y$;
	write(Y);
	unlock(Y);

Time

T_1	T_2

read_lock(Y);
read(Y);
unlock(Y);	

	read_lock(X);
	read(X);
	unlock(X);
	write_lock(Y);
	read(Y);
	$Y := X + Y$;
	write(Y);
	unlock(Y);

write_lock(X);
read(X);
$X := X + Y$;
write(X);
unlock(X);	

Transactions T_1 and T_2 (left) do not follow the two-phase locking protocol, so they allow for a non-serializable schedule (right)

Two-phase locking: positive example

T_1'

read_lock(Y);
read(Y);
write_lock(X);
unlock(Y);
read(X);
$X := X + Y$;
write(X);
unlock(X);

T_2'

read_lock(X);
read(X);
write_lock(Y);
unlock(X);
read(Y);
$Y := X + Y$;
write(Y);
unlock(Y);

Time

T_1'	T_2'

read_lock(Y);
read(Y);
write_lock(X);	

	read_lock(X);

unlock(Y);
read(X);
$X := X + Y$;
write(X);
unlock(X);	

	read(X);
	write_lock(Y);
	unlock(X);
	read(Y);
	$Y := X + Y$;
	write(Y);
	unlock(Y);

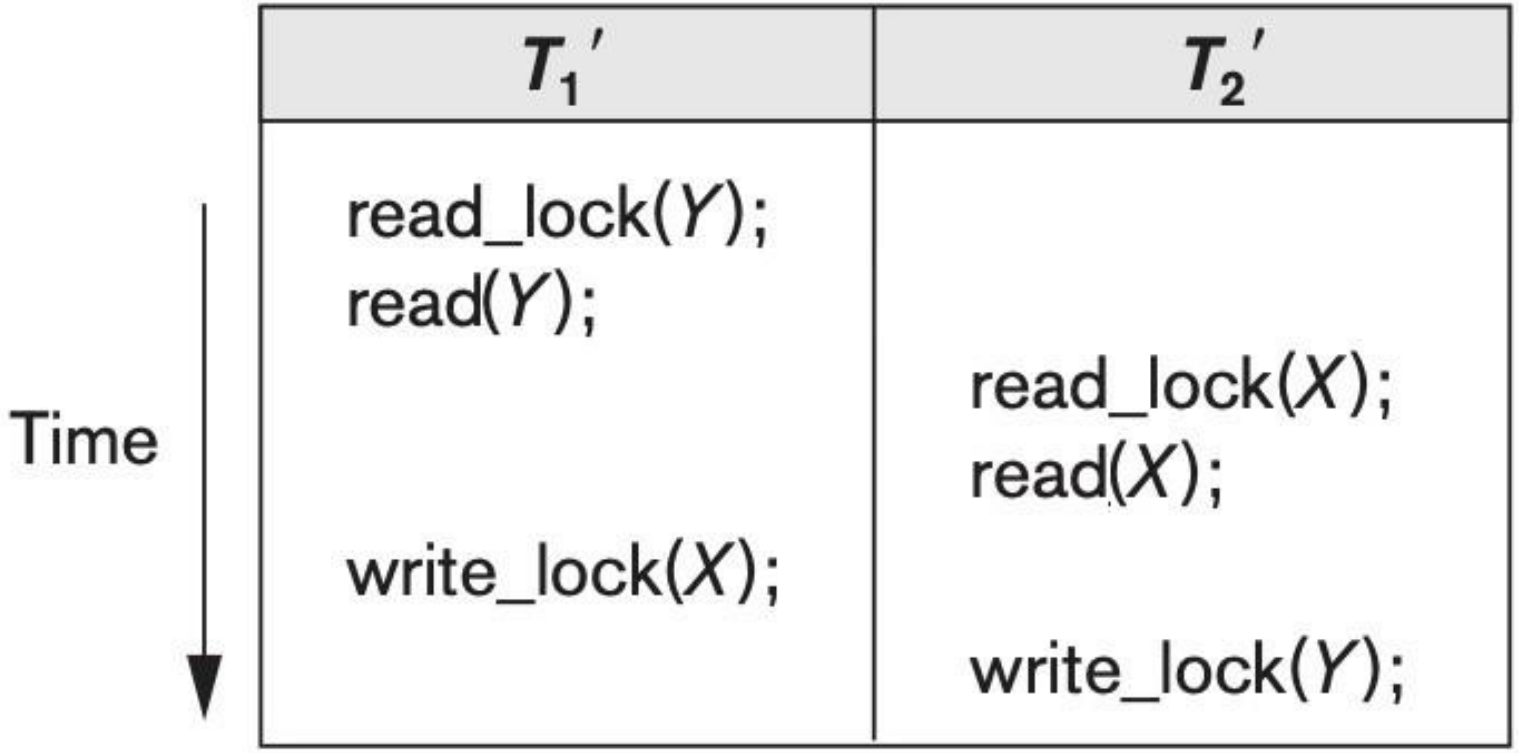
Versions T_1' and T_2' (left) follow the two-phase locking protocol, so no non-serializable schedule complies the rules

serializability by two-phase locking

- Two-phase locking guarantees serializability
  - (i.e., every allowed schedule of transactions following the rules of the two-phase locking protocol is (conflict-serializable)
- As the result, we can ***create a serializable schedule operation by operation, without knowing the transactions in full***

Drawback

- we may end up in a **deadlock**



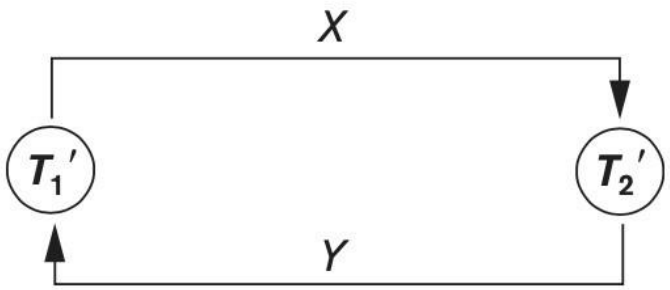
Deadlocks

Deadlock problem

- Two (or more) transactions may be stuck in a deadlock, where each transaction is waiting for another transaction, making a 'waiting cycle'

T_1'	T_2'
read_lock(Y);	
read(Y);	
	read_lock(X);
	read(X);
write_lock(X);	
	write_lock(Y);

Time



- This may happen for all types of locks,

Deadlock avoidance: Overview

- protocols that deal with deadlocks:
  - (naive)
  - deadlock prevention
  - deadlock detection
  - timeouts
- Most of protocols may roll back (i.e., abort and restart) some transactions
  - **cascadeless** (i.e., **recoverable**) due to two-phase locking (all deadlocked transactions in the growing phase, so no one have read what they have written)
  - may face and have to deal with **starvation**:
    - a transaction rolls back over and over again indefinitely

naive approaches

- conservative two-phase locking (not practical)
- OR
- locking according to a predefined order of all items
  - all items are ordered in advance (e.g., by address)
  - all locking of a transaction is in the increasing order
  - no deadlocks by construction
  - not practical due to
    - limited concurrency
    - unreasonable restrictions on transactions

Deadlock prevention protocols

- Timestamp-based (wait-die and wound-wait)
- Waiting-based (no-waiting and cautious-waiting)
  - All four **avoid** deadlocks, but often **force** transaction aborts without a real deadlock
- Deadlock prevention
  - is *not a popular* approach in practice due to **high overhead**,
  - but may be useful when transactions are **long** and **use many items**

Timestamp-based deadlock prevention

- Transaction timestamp:
  - a unique number **TS (T)** assigned to each transaction T so that
  - **TS (T′)** < **TS (T)** for each **T′** started before (for the first time)

Wait-die rule

- let transaction **T** try to lock an item that is already locked by **T′**
  - ★ if **TS (T)** < **TS (T′)** (i.e., **T** is older than **T′**), then **T** *waits* for **T′** to release
  - ★ otherwise (i.e., **T** is younger) **T** aborts (i.e., '*dies*') and restarts with the same timestamp
- The older proceeds (maybe, after waiting) and the younger restarts, so the wait-die rule guarantees no **deadlocks**, no **starvation**

Wound-wait rule

- let transaction **T** try to lock an item that is already locked by **T′**
  - if **TS (T)** < **TS (T′)** (i.e., **T** is older than **T′**), then **T′** aborts (i.e., is '*wounded*') and restarts with the same timestamp
  - otherwise (i.e., **T** is younger) **T** *waits* for **T′** to release
- The older proceeds and the younger restarts (maybe, after waiting), so the wound-wait rule guarantees no **deadlocks**, no **starvation**

Examples

- When T1 tries to lock X that is already locked by T2

- if T1 is older
  - wait-die: T1 waits
- b1, . . . , b2, . . . , l2(X), . . . , l1(X), [T1 waits] . . . , u2(X), . . .
  - wound-wait: T2 dies
- b1, . . . , b2, . . . , l2(X), . . . , l1(X), a2, . . .

- if T1 is younger
  - wait-die: T1 dies
- b2, . . . , b1, . . . , l2(X), . . . , l1(X), a1, . . .
  - wound-wait: T1 waits
- b2, . . . , b1, . . . , l2(X), . . . , l1(X), [T1 waits] . . . , u2(X), . . .

Waiting-based deadlock prevention

- **(No timestamps in these protocols)**
- Let transaction **T** try to lock an item that is already locked by **T′**
  - no-waiting rule:
    - **T** *aborts* and restarts
  - cautious-waiting rule:
    - if **T′** is not waiting for anyone, then **T** *waits* (regularly checked)
    - otherwise, **T** *aborts* and restarts
- Any of these rules guarantees **no deadlocks** (no cyclic waiting is possible)
- Special care should be taken to **avoid starvation**.
- **Easier**, but even **more unnecessary aborts**

Deadlock detection

- Deadlock happens when there is a cycle of transactions waiting for each other
- We can detect a cycle and break the cycle by aborting one transaction (the victim)
- Can be done by maintaining the wait-for graph:
  - nodes: active transactions
  - (directed) edges: transactions waiting for transactions
- A cycle in the wait-for graph means a deadlock
- May be resource-consuming to maintain
- Victim selection should be done cautiously to avoid starvation

Timeouts

- Timeout-based deadlock avoidance:
  - fix a waiting timeout period in advance
  - a transaction waits for the timeout period (*at most*) after this abort and restart
- Simple and easy to maintain
- No guarantees
  - (unknown overhead, possible starvation)

Timestamps

Timestamp-based concurrency control

- We can use timestamps for concurrency control without locking:
  - Run transactions until something goes wrong (i.e., a conflict is detected)
  - Roll back one of the conflict transactions based on their timestamps
    - roll back is *abort and restart* with a **new** timestamp

Timestamp-based concurrency control

- Timestamp-based concurrency control protocols:
  - all ensure serializability
    - transactions arranged by their timestamps
  - may not guarantee recoverability
  - may not guarantee no starvation
- Several timestamp-based protocols
  - basic: ensures serializability, but not recoverability
  - strict: ensures serializability and strictness (so recoverability)
  - basic/strict with Thomas write rule: less roll-backs

Item timestamp variables

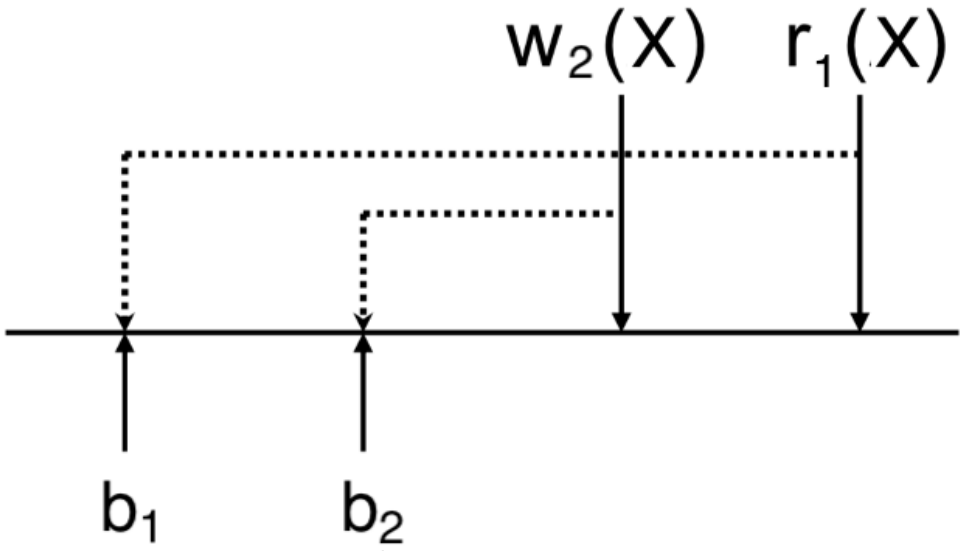
- Besides the timestamp of a transaction , the concurrency control subsystem maintains two variables for each item X :
  - read timestamp $ReadTS(X)$:
    - the timestamp of the youngest transaction that has read X
  - write timestamp $WriteTS(X)$:
    - the timestamp of the youngest transaction that has written X
- Maintained in appropriate timestamp tables
  - (similar to the lock table in lock-based protocols)
- (All initialised by 0)

Basic timestamp-ordering protocol

- If there is a conflict that violates the order (of the serial schedule imposed by the timestamps),
- then roll back the transaction of the second operation of the conflict

Read-write conflict 1

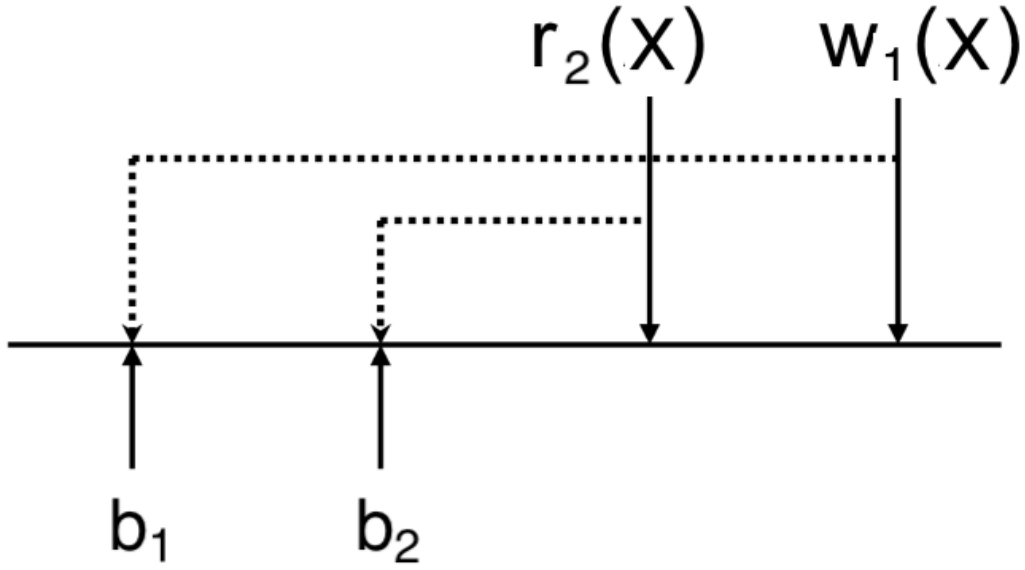
- a transaction may try to **read** data item X **too late**



- Should abort and restart **T1**

Read-write conflict 2

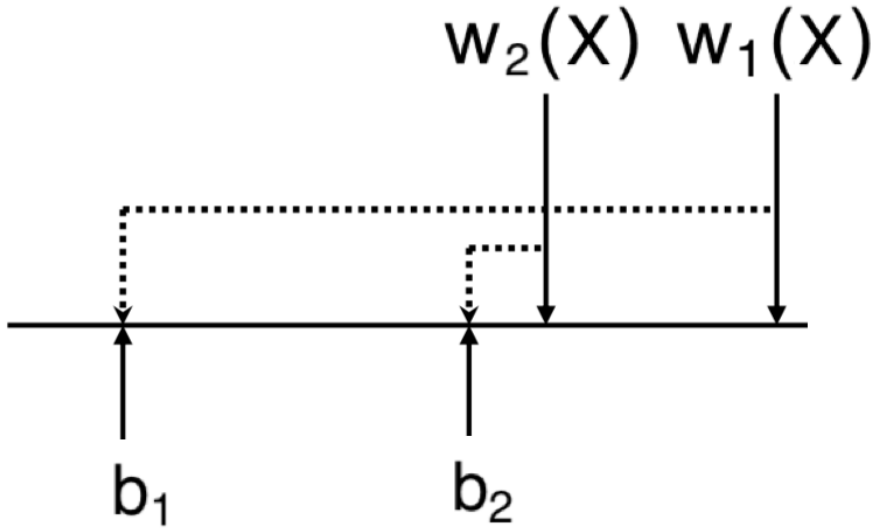
- a transaction may try to **write** data item X **too late**



- Should abort and restart **T1**

Write-write conflict

- a transaction may try to **write** data item X **too late**



- Should abort and restart **T1**

Basic timestamp-ordering Rules

- **Rule 1. When transaction *T* issues read(*X*):**
  - if *WriteTS* (*X*) > *TS* (*T*) (i.e., someone younger written *X*),
  - then roll-back *T* (i.e., abort and restart *T* with a new timestamp)
  - otherwise, execute the rest of read(*X*) and set *ReadTS* (*X*) to max(*ReadTS* (*X*), *TS* (*T*))
- **Rule 2. When transaction *T* issues write(*X*):**
  - if *ReadTS* (*X*) > *TS* (*T*) or *WriteTS* (*X*) > *TS* (*T*) (i.e., someone younger accessed *X*),
  - then roll-back *T* (i.e., abort and restart *T* with a new timestamp)
  - otherwise, execute the rest of write(*X*) and set *WriteTS* (*X*) to *TS* (*T*)

Basic timestamp-ordering properties

- Ensures serializable schedules, but they
  - may be not recoverable,
    - (ensured by strict version)
  - may have cascading roll-backs,
    - (avoided by strict version)
  - may have starving transactions

Strict timestamp-ordering protocol

- Idea
  - In the rules the execution is postponed (using e.g., locking) until the pervious write is committed (so ensuring strict schedules)

Rule 1

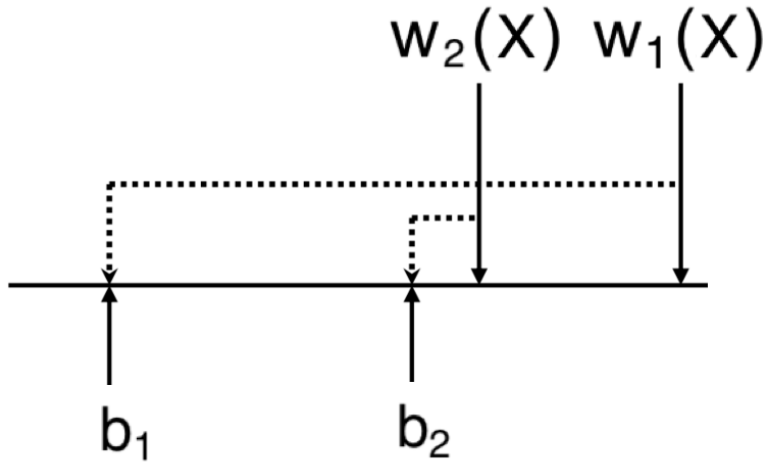
- When transaction T issues read(X):
  - if $WriteTS(X) > TS(T)$ then roll-back T
  - otherwise
    - ★ wait until the transaction T' such that
      - $WriteTS(X) = TS(T')$ is committed (if there is such)
    - ★ execute the rest of read(X) and
      - set $ReadTS(X)$ to max($ReadTS(X)$, $TS(T)$)

Rule 2

- When transaction T issues write(X):
  - if $ReadTS(X) > TS(T)$ or $WriteTS(X) > TS(T)$ then roll-back T
  - otherwise
    - ★ wait until the transaction T' such that
      - $WriteTS(X) = TS(T')$ is committed (if there is such)
    - ★ execute the rest of write(X) and set $WriteTS(X)$ to $TS(T)$

Thomas writing rule

- Improvement (less roll-backs) for both basic and strict protocols in case of write-write conflict



- Rule 1. When transaction *T* issues read(*X*): . . . (same)
- Rule 2. When transaction *T* issues write(*X*):
  - ★ if *ReadTS* (*X*) > *TS* (*T*) then roll-back *T*
  - ★ if *WriteTS* (*X*) > *TS* (*T*) ignore the operation and proceed
  - ★ otherwise . . . (same, i.e., possibly wait, execute, update)
- ***Same guarantees as the versions without the improvement***

Multi-versioning

Multiversioning concurrency control

- Multi-versioning idea:
  - several versions of the (value of the) same item are kept by the system
  - transactions are the same as usual (in particular, they request reads/writes for items, not versions)
  - the versions of items are managed by the concurrency control subsystem
- Pros: increase concurrency
- Cons: more storage is needed to maintain multiple versions

Concurrency with Multiple granularity

Data item granularity

- Granularity: the size of data items
  - field value of a record (fine)
  - record
  - disk block
  - table file
  - whole database (coarse)
- The tradeoff:
  - more coarse allows for lower degree of concurrency
  - more fine leads to more overhead (lock management, etc.)
- Best item size depends on transaction type:
  - if a transaction accesses few records, a record as item is good
  - if a transaction accesses many records, a block as item is good
- It may be nice to have different granularity for different transactions

THANK YOU