Database Systems II

Lecture 10

Recoverability, serializability, and Concurrency Control

Addison-Wesley
is an imprint of
PEARSON

What is Schedule

- **Schedule** is a sequence of the operations by a set of *concurrent transactions* that *preserves the order* of the operations in each of the individual transactions.

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

Characterizing Schedules based on Recoverability

- It is important to characterize the types of schedules for which recovery is possible, as well as those for which recovery is relatively simple.
- These characterizations do not actually provide the recovery algorithm
- They only attempt to theoretically characterize the different types of schedules
- It is important to ensure that once a transaction T is committed, it should never be rolled back. This ensures that the durability property of transactions is not violated

Recoverable/Non-recoverable Schedules

- **Non-recoverable Schedule:**

  A schedule where a committed transaction may have to be rolled back during recovery.

  To preserve the **Durability** properties, non-recoverable schedules *should not be allowed*.

- **Recoverable Schedule:**

  A schedule *S* is **recoverable** if no transaction *T* in *S* commits until all transactions *T'* that have written an item that *T* reads have committed.

Recoverable/Non-Recoverable Schedules

- **Example:** Schedule A below is **non-recoverable** because T2 reads the value of X that was written by T1, but then T2 commits before T1 commits or aborts

Schedule A

T1	T2
R(x)	
W(X)	
	R(X)
	W(X)
	commit
R(Y)	
W(Y)	
abort	

Recoverable/Non-Recoverable Schedules(cont.)

- To make the previous **schedule recoverable**, the commit of T2 (c2) must be delayed until T1 either commits, or aborts (Schedule B)
- If T1 commits, T2 can commit
- If T1 aborts, T2 must also abort because it reads a value that was written by T1; this value must be undone (reset to its old value) when T1 is aborted

T1	T2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
Commit or abort	
	Commit or abort

Schedule B

Characterizing Schedules based on Recoverability

Recoverable schedules can be further refined**:**

- **Cascadeless schedule:** A schedule in which a transaction T2 cannot read an item X until the transaction T1 that last wrote X has committed.
- The set of cascadeless schedules is a *subset of* the set of recoverable schedules.

Schedules requiring cascaded rollback: A schedule in which an uncommitted transaction T2 that read an item that was written by a failed transaction T1 must be rolled back.

Characterizing Schedules Based on Recoverability

- **Example:** Schedule B below is **not cascadeless** because T2 reads the value of X that was written by T1 before T1 commits
- If T1 aborts (fails), T2 must also be aborted (rolled back) resulting in *cascading rollback*
- To make it **cascadeless**, the r2(X) of T2 must be delayed until T1 commits (or aborts and rolls back the value of X to its previous value) – see Schedule C

- Schedule B: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y); c1 (or a1);

- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); ...

Characterizing Schedules based on Recoverability

Cascadeless schedules can be further refined:

- **Strict schedule:** A schedule in which a transaction T2 can neither read *nor write* an item X until the transaction T1 that last wrote X has committed.
- The set of strict schedules is a *subset of* the set of cascadeless schedules.
- If *blind writes* are not allowed, all cascadeless schedules are also strict

Blind write: A write operation w2(X) that is not preceded by a read r2(X).

Characterizing Schedules Based on Recoverability

- **Example:** Schedule C below is **cascadeless** and also **strict** (because it has no blind writes)
- Schedule D is cascadeless, but not strict (because of the blind write w3(X), which writes the value of X before T1 commits)
- To make it strict, w3(X) must be delayed until after T1 commits – see Schedule E

- Schedule C: r1(X); w1(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule D: r1(X); w1(X); w3(X); r1(Y); w1(Y); c1; r2(X); w2(X); …
- Schedule E: r1(X); w1(X); r1(Y); w1(Y); c1; w3(X); c3; r2(X); w2(X);

Characterizing Schedules Based on Recoverability

Summary:

- Many schedules can exist for a set of transactions
- The set of all possible schedules can be partitioned into two subsets: ***recoverable*** and ***non-recoverable***
- A subset of the recoverable schedules are ***cascadeless***
- If *blind writes are not allowed*, the set of cascadeless schedules is ***strict*** schedules

Serializability

Characterizing Schedules based on Serializability

- Among the large set of possible schedules, we want to characterize which schedules are guaranteed to give a correct result.
- The **consistency preservation** property of the ACID properties states that: each transaction if executed on its own (from start to finish) will transform a consistent state of the database into another consistent state.
- Hence, *each transaction is correct on its own.*

Serial/Non-Serial Schedules

Serial schedule:

- A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions. Otherwise, the schedule is called *non-serial*.

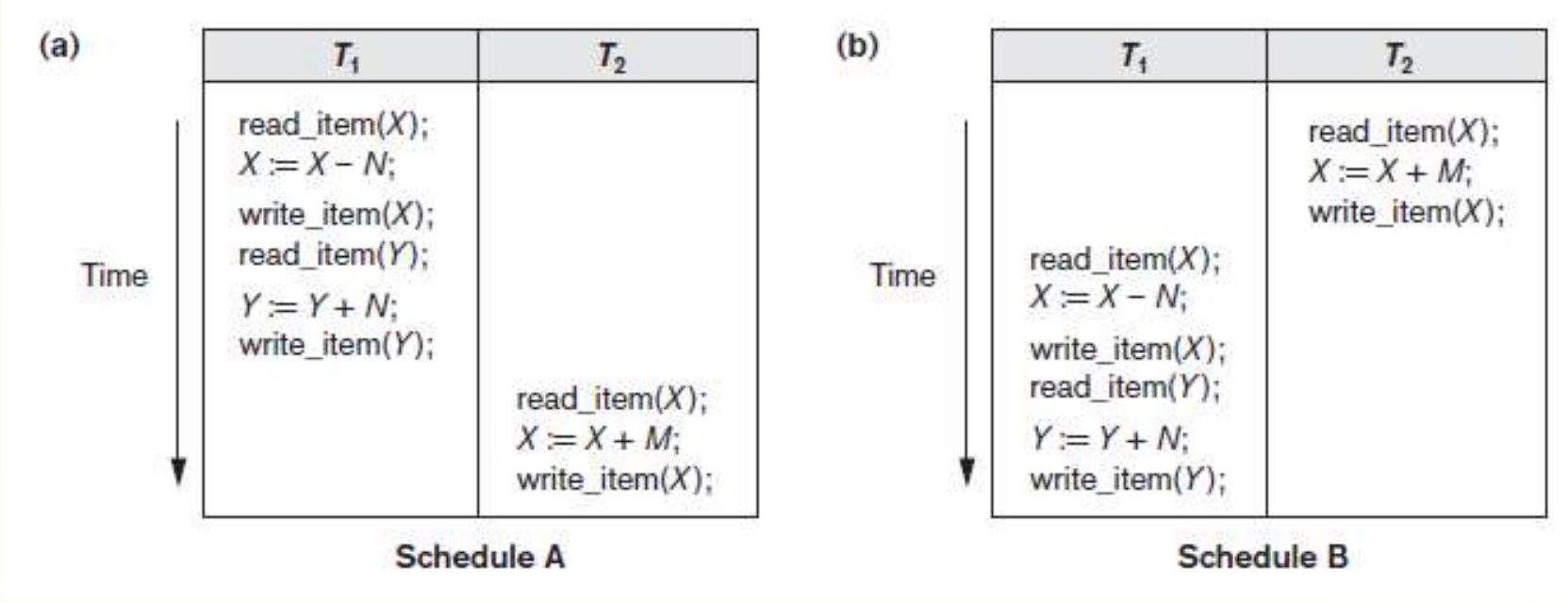
Non-serial schedule:

- A schedule where the operations from a set of concurrent transactions are interleaved

Based on the consistency preservation property, *any serial schedule will produce a correct result*

Example of Serial Schedule

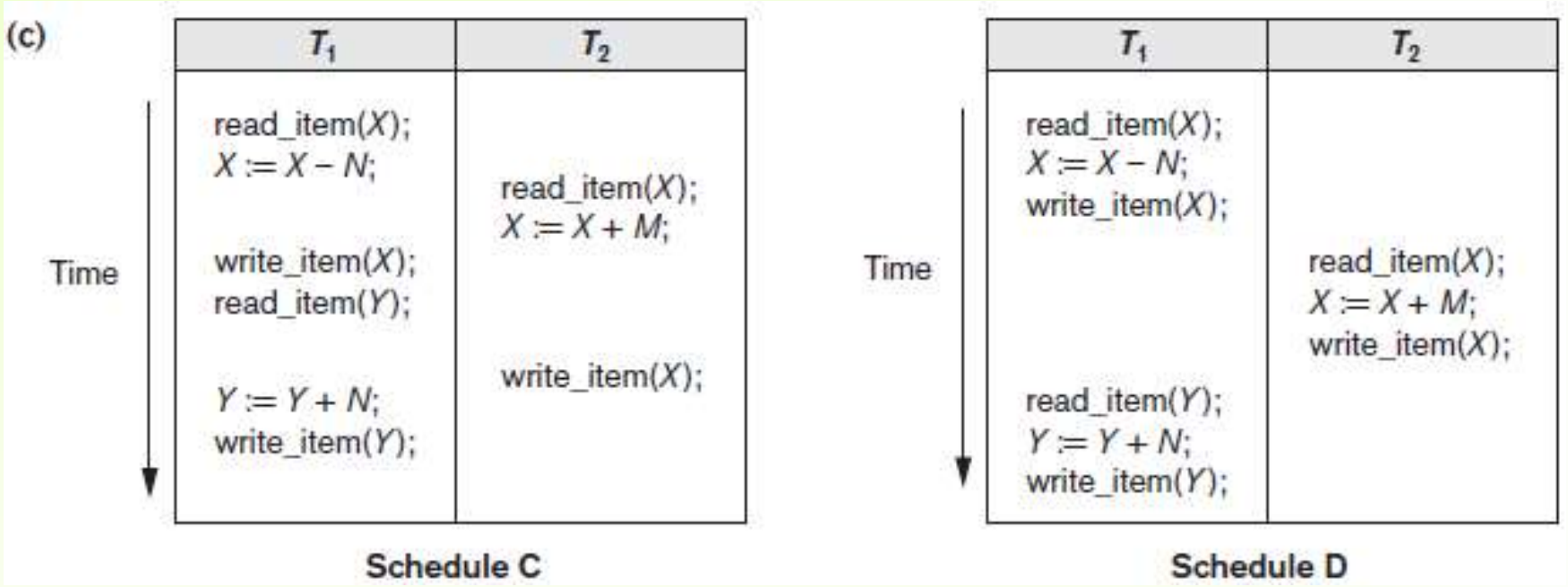
Suppose that X= 90 and Y= 90, M=2 and N=3



*Final result X= 89 and Y= 93 for both schedules

Example of Non-Serial Schedule

Suppose that X= 90 and Y= 90, M=2 and N=3



Schedule C produces X= 92 and Y= 93 while schedule D produces right result

Serializable Schedule

- If a set of transactions executes concurrently, we say that the (non-serial) schedule is **Serailizable** if it *produces the same results as some serial execution*.
- Non-serializable schedules are likely to result in inconsistent databases state.
- Saying that a *non-serial* schedule *S is serializable,* is equivalent to saying that it is correct, because it is equivalent to a serial schedule, which is considered correct.

Concurrency Control

Purpose of Concurrency Control

- To ensure that the Isolation Property is maintained while allowing transactions to execute concurrently (outcome of concurrent transactions should appear as though they were executed in isolation).
- To preserve database consistency by ensuring that the schedules of executing transactions are serializable.
- To resolve read-write and write-write conflicts among transactions.
- Locking is one of the main techniques used to control concurrent execution of transactions.

What is a Locks

- **A lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.

  – A process before read/write ➔ requests the scheduler to grant a lock

  – Upon finishing read/write ➔ the lock is released

- Locks are used as a means of synchronizing the access by concurrent transactions to the database items.

Locking Techniques: Essential components

Lock Manager: Subsystem of DBMS that manages locks on data items.

Lock table: Lock manager uses it to store information about *locked data items*, such as: data item id, transaction id, lock mode, list of waiting transaction ids, etc. One simple way to implement a lock table is through linked list.

How Locks prevent consistency problems

- Lost update and inconsistent retrieval:
- Causes: are caused by the conflict between ri(x) and wj(x)
  - Two transactions read a value and use it to compute new value
- Prevention:
  - Delay the reads of later transactions until the earlier ones have completed

Disadvantage of Locking

Deadlocks

Types of Locks

1- Binary Lock

- It is simple, but it is also too restrictive for database concurrency control purposes, *and so are not used in practice*
- Can have two states or values: locked and unlocked or 1 and 0
- A distinct lock is associated with each database item *X.* If the value of the lock on X is 1, item X cannot be accessed by a database operation that requests the item. If the value of the lock on X is 0 the item can be accessed when requested

Types of Locks

1- Binary Lock

- Two operations, lock_item and unlock_item, are used with binary locking

- Transaction requests access to item X has two options:
  - If Lock(X)=1 then transaction has to wait and is put on a waiting queue
  - If Lock(X) =0 then transaction is granted access to the item and the value of Lock(X) will be 1

- Binary lock enforces mutual exclusion on the data item

Types of Locks

1- Binary Lock

```
lock_item(X):
B:  if LOCK(X) = 0              (* item is unlocked *)
        then LOCK(X) ←1         (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 22.1
Lock and unlock operations for binary locks.

Types of Locks

2- Shared/Exclusive Locks

- For database purposes, binary locks are not sufficient:
- Two locks modes are needed:
- **Shared mode:** Read lock (X). *Several transactions* can hold shared lock on X (because read operations are not conflicting).

- **Exclusive mode:** Write lock (X). *Only one write* lock on X can exist at any time on an item X. (No read or write locks on X by other transactions can exist).

Types of Locks

2- Shared/Exclusive Locks

- **Three operations are now needed:**
  - read_lock(X): transaction T requests a read (shared) lock on item X
  - write_lock(X): transaction T requests a write (exclusive) lock on item X
  - unlock(X): transaction T unlocks an item that it holds a lock on (shared or exclusive)
- Transaction can be blocked (forced to wait) if the item is held by other transactions **in write lock mode**

Types of Locks

2- Shared/Exclusive Locks (how to implement)

- **Lock table contains four fields:**
  - Data_item_name, Lock, No_of_reads, Locking_transactions
  - Lock is either read lock or write lock
  - Locking transactions value is one transaction in case of write lock and contains multiple transactions in case of read lock
  - No_of_reads is the number of transactions reading the data item

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                   no_of_reads(X) ← 1
                   end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
             wait (until LOCK(X) = "unlocked"
                 and the lock manager wakes up the transaction);
             go to B
             end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
             wait (until LOCK(X) = "unlocked"
                 and the lock manager wakes up the transaction);
             go to B
             end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                   wakeup one of the waiting transactions, if any
                   end
    else it LOCK(X) = "read-locked"
        then begin
                   no_of_reads(X) ← no_of_reads(X) –1;
                   if no_of_reads(X) = 0
                       then begin LOCK(X) = "unlocked";
                                  wakeup one of the waiting transactions, if any
                                  end
                   end;
```

Figure 22.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

Types of Locks

2- Shared/Exclusive Locks: Rules for locking

- Transaction must request appropriate lock on a data item X before it reads or writes X.
- If T holds a write (exclusive) lock on X, it can both read and write X.
- If T holds a read lock on X, it can only read X.
- A transaction *T* must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.

Types of Locks (Cont.)

3- Lock Conversion

Lock upgrade: existing read lock to write lock

if *Ti* holds a read-lock on X, and no other *Tj* holds a read-lock on X , then

    it is possible to convert (**upgrade**) read-lock(X) to write-lock(X)

else

    force *Ti* to wait until all other transactions *Tj* that hold read locks on X release their locks

Lock downgrade: existing write lock to read lock

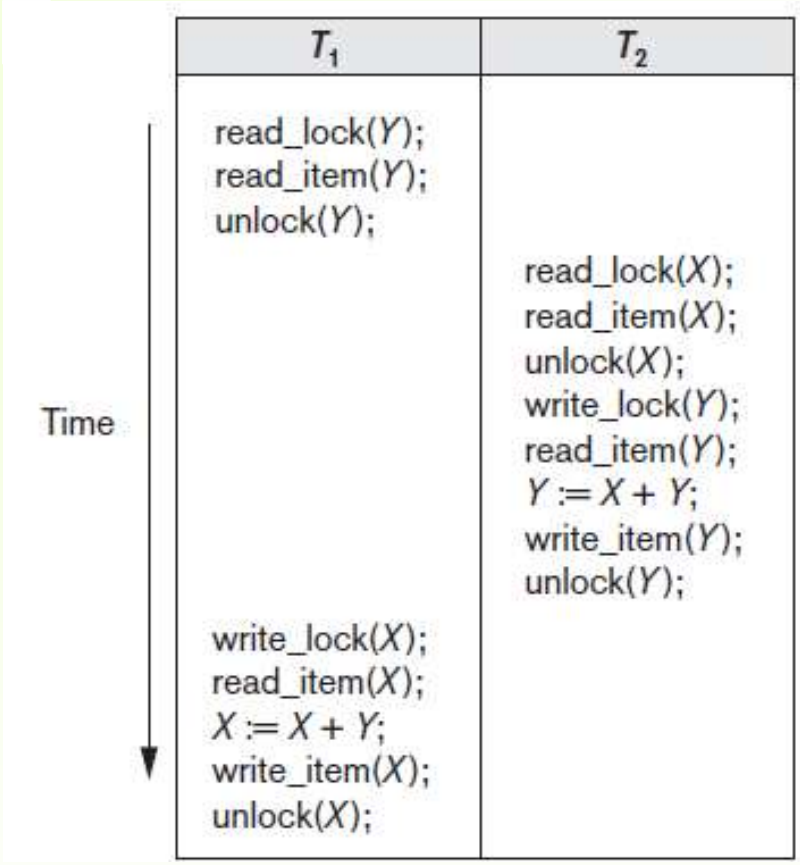
if *Ti* holds a write-lock on X

(*this implies that no other transaction can have any lock on X*)
then it is possible to convert (**downgrade**) write-lock(X) to read-lock(X)

2Phase Locking Concurrency Control

Using locks doesn't guarantee serializability in its own

- To guarantee serializability, an *additional protocol* concerning the positioning of locking and unlocking operations in every transaction must be followed.
- The Two-Phase Locking (2PL) is the best-known protocol.



The items Y in T_1 and X in T_2 were unlocked too early.

2Phase Locking Concurrency Control

- A transaction is said to follow the two-phase locking protocol if all locking operations (read_lock, write_lock) precede the first unlock operation in the transaction.
- **Each transaction should have two phases**: (a) Locking (Growing) phase, and (b) Unlocking (Shrinking) Phase.
- **Locking (Growing) Phase**: A transaction applies locks (read or write) on desired data items one at a time. Can also try to upgrade a lock.
- **Unlocking (Shrinking) Phase**: A transaction unlocks its locked data items one at a time. Can also downgrade a lock.

2PL Concurrency Control

- **Requirement:** For a transaction, these two phases must be mutually exclusive, that is, during locking phase no unlocking or downgrading of locks can occur, and during unlocking phase no new locking or upgrading operations are allowed.
- When transaction starts executing, it is in the **locking phase,** and it can request locks on new items or upgrade locks. A transaction may be blocked (forced to wait) if a lock request is not granted.
- Once the transaction unlocks an item (or downgrades a lock), it starts its **shrinking phase** and can no longer upgrade locks or request new locks.

Theorem: **If *every transaction* in a schedule follows the 2PL rules, the schedule must be serializable**

2PL Concurrency Control

Example:

T1	T2
	Begin transaction
Begin transaction	Write_lock (X)
Write_lock(X)	Read (X)
Wait	Write_lock(Y)
Wait	Read (Y)
Wait	Y=X+Y
Wait	Unlock (X)
Read(X)	Write (Y)
	Unlock (Y)

Problems of 2 Phase Lock

Deadlock

A dead end that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.

T1'	T2'
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
(waits for X)	
	write_lock (Y);
	(waits for Y)

Deadlock (T1' and T2')

Dealing with Deadlock

Timeouts

- Transaction that requests a lock will wait for only a system-defined period of time.
- If the lock has not been granted within this period, the lock request times out.
- In this case, the DBMS assumes the transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction.
- This is a very simple and practical solution to deadlock prevention and is used by several commercial DBMSs.