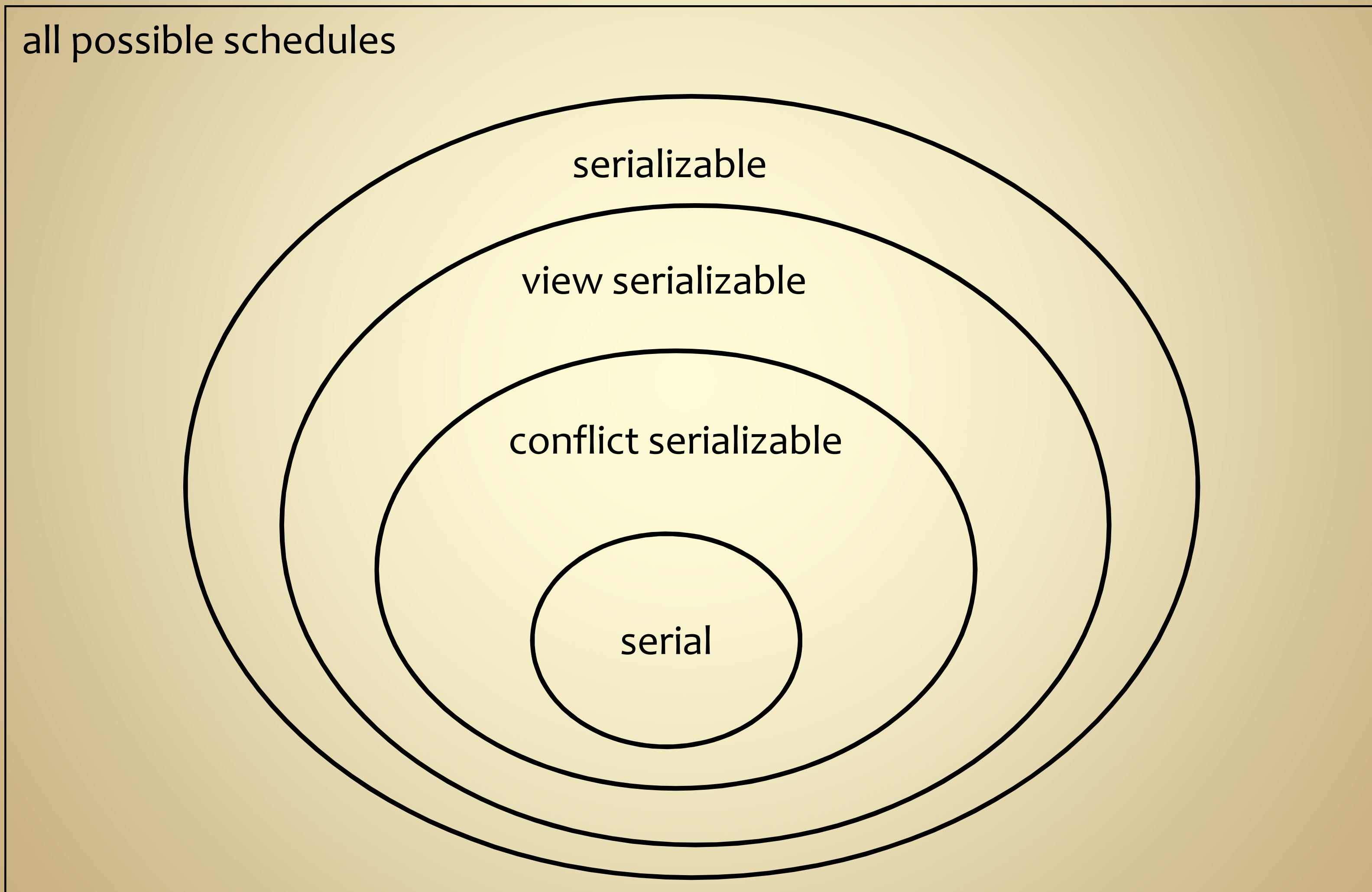
COMP163

Database Management Systems

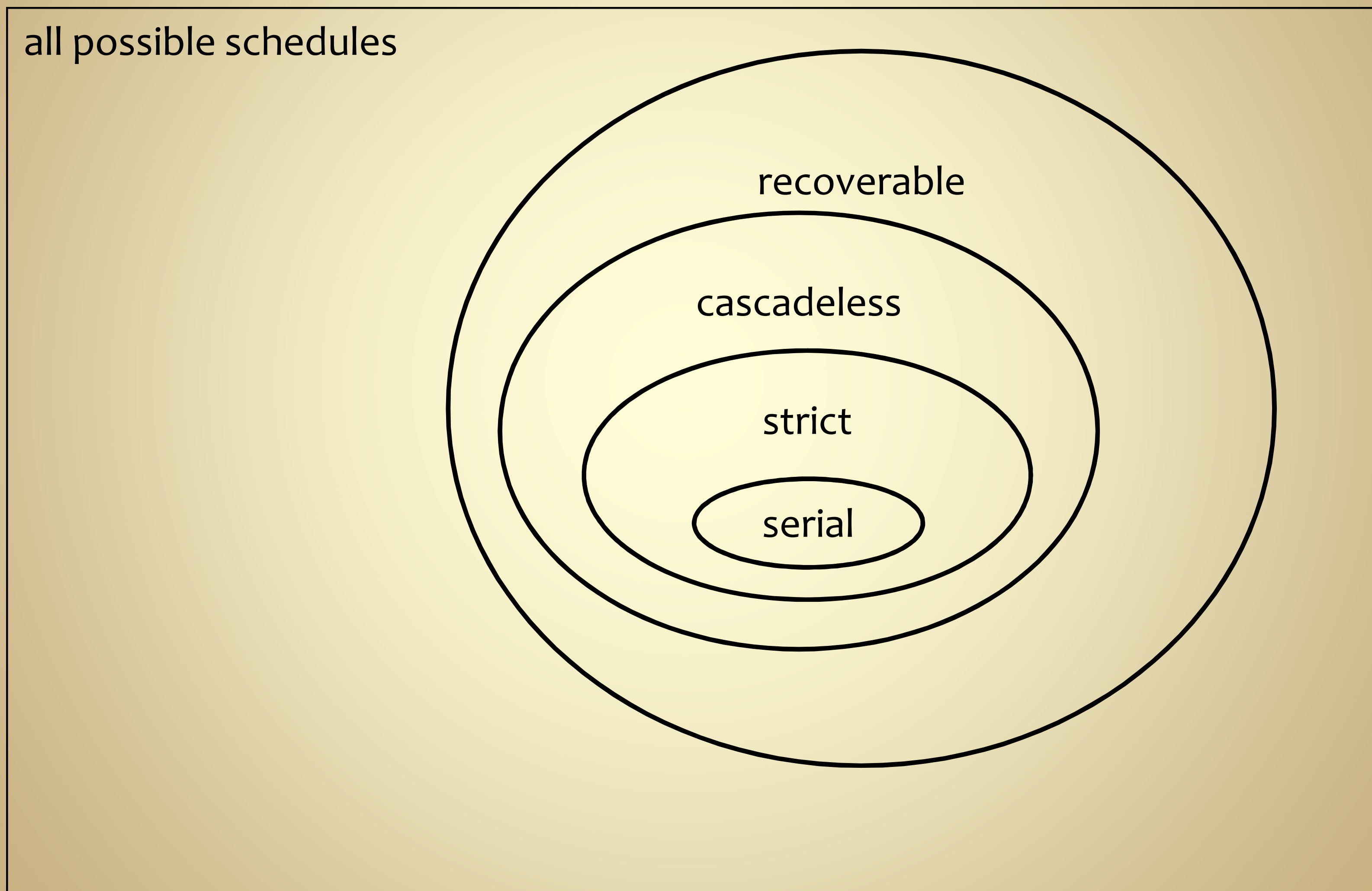
Concurrency Control: Locking
Chapter 22

Serializability Classes



REVIEW

Recoverability Classes



REVIEW

Concurrency Control

- **Transaction theory** classifies possible schedules in terms of recoverability and correctness
- **Concurrency control** implements mechanisms to achieve specific policies
- Pessimistic CC:
  - prevent unwanted schedules from occurring
  - may reduce concurrency and stall transactions
- Optimistic CC:
  - allow any schedule
  - check at commit time and abort transactions contributing to unwanted schedules

REVIEW

Locking

CC: Locking Protocols

- **Locking** is an operation that secures
  - permission to ***read***, and/or
    permission to ***write*** a data item for a transaction
  - Example:
    - Lock(X): Data item X is locked on behalf of the requesting transaction

- **Unlocking** is an operation that removes these permissions from the data item.
  - Example:
    - Unlock(X): Data item X is made available to all other transactions

- Lock and Unlock are ***atomic*** operations
- Locking implements pessimistic CC

SQL Isolation Levels

isolation level	prevents	locking
READ UNCOMMITTED		all locks released immediately following SQL statement execution
READ COMMITTED	dirty reads	read locks released immediately write locks held until end of transaction
REPEATABLE READ	dirty reads non-repeatable reads	all locks held until end of transaction (strict 2PL)
SERIALIZABLE	dirty reads non-repeatable reads phantom records	requires index or table locks to prevent phantom reads

Recoverability Classes

- Recoverable schedule test:
  - ***no transaction T commits***
    until all transactions that wrote something
    that T reads have committed
- Cascadeless test:
  - ***every transaction only reads things***
    written by committed transactions
- Strict schedule test:
  - ***no transaction can read or write anything***
    that was written by an uncompleted transaction

REVIEW

Two Types of Locks

- Two locks modes:
  - shared (read)
  - exclusive (write)
- Shared lock: s(X)
  - Multiple transactions can hold a shared lock on X
  - No exclusive lock can be applied on X while a shared lock is held on X
- Exclusive lock: x(X)
  - Only one exclusive lock on X can exist at any time
  - No shared lock can be applied on X when an exclusive lock is held on X

Lock Granting

locks held by other transactions

requested lock	none	s(X)	x(X)
s(X)	grant	grant	wait
x(X)	grant	wait	wait

Well-formed Transactions

- Locking assumes that all transactions are well-formed

- A transaction is well-formed if:
  - It properly locks a data item before it reads or writes to it
  - It does not lock an item locked by another transaction
  - It does not unlock an item that it does not hold a lock on

- More simply:
  Well-formed transactions obey locking rules

Basic Lock/Unlock Algorithm

```
Lock(X):
START:
   if lock(X) = 0 then
      lock(X) ← 1
   else
      wait until (lock(X) = 0)
      goto START

Unlock(X):
   lock(X) ← 0 (*unlock the item*)
   wake any transaction waiting for lock on X
```

Shared-Lock Requests

```
START:
   if lock(X) = "unlocked" then
      lock(X) ← "shared-lock"
      no_of_reads(X) ← 1

   else if lock(X) = "shared-lock" then
      no_of_reads(X) ← no_of_reads(X) + 1

   else (* must be an exclusive lock *)
      wait until (LOCK(X) = "unlocked")
      go to START
```

Exclusive-Lock Requests

```
START:
   if lock(X) = "unlocked"
       lock(X) ← "exclusive-lock"
   else
      wait until (lock(X) = "unlocked")
      go to START
```

Unlocking

```
if LOCK(X) = "exclusive-lock"
   LOCK (X) ← "unlocked"
    wake up a waiting transactions (if any)

else if LOCK(X) ← "shared-lock"
   no_of_reads(X) ← no_of_reads(X)-1
   if no_of_reads(X) = 0
      LOCK(X) = "unlocked"
      wake up a waiting transactions (if any)
```

Lock Conversions

- Lock upgrade: convert shared lock to exclusive lock

  if T has the only shared lock on X
      convert shared-lock(X) to exclusive-lock(X)
  else
      force T to wait until all other transactions unlock X

- Lock downgrade: convert exclusive lock to shared lock

  if T has an exclusive-lock(X)
      convert exclusive-lock(X) to shared-lock(X)

Two Phase Locking

Two-Phase Locking

- **Two Phases:**
  - Locking (Growing)
  - Unlocking (Shrinking)
- **Locking (Growing) Phase:**
  - A transaction applies locks (read or write) on desired data items one at a time
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time
- **Requirement:**
  - Within any transaction these two phases must be mutually exclusive – once you start unlocking, you cannot request any more locks

Two-Phase Locking

- Locking itself does not imply serializability
- 2PL guarantees serializability
  - improper ordering of operations is prevented
  - if 2PL is enforced, there is no need to test schedules for serializability
- 2PL limits concurrency
  - locks may need to be held longer than needed
- Basic 2PL may cause deadlock

Locking Example

T1		T2	
read_lock (Y)	s1(Y)	read_lock (X)	s2(X)
read_item (Y)	r1(Y)	read_item (X)	r2(X)
unlock (Y)	u1(Y)	unlock (X)	u2(X)
write_lock (X)	x1(X)	write_lock (Y)	x2(Y)
read_item (X)	r1(X)	read_item (Y)	r2(Y)
X:=X+Y		Y:=X+Y	
write_item (X)	w1(X)	write_item (Y)	w2(Y)
unlock (X)	u1(X)	unlock (Y)	u2(Y)

Initial values: X=20; Y=30

Result of serial execution, T1 followed by T2: X=50, Y=80

Result of serial execution, T2 followed by T1: X=70, Y=50

Locking Example

T1		T2	
read_lock (Y)	s1(Y)	read_lock (X)	s2(X)
read_item (Y)	r1(Y)	read_item (X)	r2(X)
unlock (Y)	u1(Y)	unlock (X)	u2(X)
write_lock (X)	x1(X)	write_lock (Y)	x2(Y)
read_item (X)	r1(X)	read_item (Y)	r2(Y)
X:=X+Y		Y:=X+Y	
write_item (X)	w1(X)	write_item (Y)	w2(Y)
unlock (X)	u1(X)	unlock (Y)	u2(Y)

Both transactions obey basic locking protocols, since they hold appropriate locks when reading or writing data items.

Neither transaction obeys 2PL.

Locking Example

T1	T2
s1(Y)	
r1(Y)	
u1(Y)	
	s2(X)
	r2(X)
	u2(X)
	x2 (Y)
	r2(Y)
	Y:=X+Y
	w2(Y)
	u2(Y)
x1(X)	
r1(X)	
X:=X+Y	
w1(X)	
u1(X)	

Time

Result: X=50; Y=50

This schedule is legal in that it obeys locking rules, but it is not serializable (and not correct)

violates two-phase policy

2PL Example

T1	**T2**	
s1(Y)	s2(X)	
r1 (Y)	r2(X)	growing phase
x1 (X)	x2(Y)	
u1(Y)	u2(X)	
r1(X)	r2(Y)	shrinking phase
X:=X+Y	Y:=X+Y	
w1(X)	w2(Y)	
u1(X)	u2(Y)	

Both transactions obey 2PL.
It is not possible to interleave them in a manner that results in a non-serializable schedule.

Basic 2PL

- Basic 2PL requires that no locks are requested after the first unlock
- Guarantees serializability
  - transactions that request operations that violate serializability are delayed while waiting on locks
- Reduces concurrency, since locks must be held until all needed locks have been acquired
- May cause deadlock

Conservative 2PL

- Conservative 2PL requires that
  all locks must be acquired at start of transaction

- Prevents deadlock, since all locks
  are acquired as a block
  - No transaction can be waiting on one lock
    while it holds another lock

- Further restricts concurrency,
  since transaction must request strongest lock
  that *might* be needed

Strict 2PL

- Strict 2PL requires that
  all locks must be held until end of transaction
- Deadlock is possible
- Guarantees strict schedules
- May require holding locks longer than necessary
- Most commonly used algorithm

Deadlock

2PL: Deadlock

T1	**T2**
s1(Y)	s2(X)
r1 (Y)	r2(X)
x1 (X)	x2(Y)
u1(Y)	u2(X)
r1(X)	r2(Y)
X:=X+Y	Y:=X+Y
w1(X)	w2(Y)
u1(X)	u2(Y)

T1 cannot proceed until T2 releases lock on X.
T2 cannot proceed until T1 releases lock on Y.
→ DEADLOCK

Conservative 2PL: ~~Deadlock~~

<u>T1</u>	**<u>T2</u>**
s1(Y), x1 (X)	s2(X), x2(Y)
r1 (Y)	r2(X)
u1(Y)	u2(X)
r1(X)	r2(Y)
X:=X+Y	Y:=X+Y
w1(X)	w2(Y)
u1(X)	u2(Y)

In this example, the only possible schedules are serial schedules.

Locks must be acquired as a unit at beginning of transaction.
Transaction cannot be holding locks while waiting on locks.
Deadlock is not possible.

Conservative 2PL: ~~Deadlock~~

T1	**T2**
s1(Y), x1(Z)	
r1 (Y)	
u1(Y)	s2(X), x2(Y)
r1(Z)	r2(X)
Z:=Z+Y	u2(X)
w1(Z)	r2(Y)
u1(Z)	Y:=X+Y
	w2(Y)
	u2(Y)

T2 can proceed as soon as T1 releases lock on Y.

Concurrency is still possible under conservative 2PL.

Deadlock: Detection/Resolution

- Let deadlocks happen, then resolve the problem

- **Wait-for graph**
  - scheduler maintains a *wait-for graph*
    - arc from Tx to Ty indicates Tx is waiting for a lock held by Ty
  - when a transaction is blocked, it is added to the graph
  - a cycle in the wait-for-graph indicates deadlock
  - one transaction involved in the cycle is selected (victim) and rolled-back

- **Timeout**
  - abort any transaction that has been waiting for some set amount of time
  - simple solution, but may be abort a transaction that could eventually proceed

Deadlock: Prevention

- **Locking policy**
  - Implement a CC policy that never allows deadlock to occur
  - Example: conservative 2PL

- **Waits-for cycle avoidance**
  - Use wait-for graph, but do not allow cycles to occur
  - Example: any transaction that would create a cycle is aborted
  - Other algorithms use timestamps to chose victim

Deadlock Victim Selection

- T1 tries to lock X, T2 holds lock on X

  - **wait-die:**
    if T1 is older than T2, T1 waits
    otherwise, T1 aborts
  - **wound-wait:**
    if T1 is older than T2, T2 aborts,
    otherwise, T1 waits
  - **no-waiting:**
    T1 aborts
  - **cautious waiting:**
    if T2 is waiting, T1 aborts,
    otherwise T1 waits

Starvation

- **Starvation**

  - A particular transaction consistently waits or gets restarted and never gets a chance to complete
  - Caused by deadlock victim selection policy
  - Inherent in all priority based scheduling mechanisms

  - Example: Wound-Wait
    a younger transaction may always be aborted
    by a long running older transaction,

Livelock

- States of the processes involved are changing, but no process is progressing
- example: two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress
- Livelock can result from deadlock detection/recovery If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered.
  - avoided by ensuring that only one process takes action

source: Wikipedia

SQL Isolation Levels

isolation level	prevents	locking
READ UNCOMMITTED		all locks released immediately following SQL statement execution
READ COMMITTED	dirty reads	read locks released immediately write locks held until end of transaction
REPEATABLE READ	dirty reads non-repeatable reads	all locks held until end of transaction (strict 2PL)
SERIALIZABLE	dirty reads non-repeatable reads phantom records	requires index or table locks to prevent phantom reads