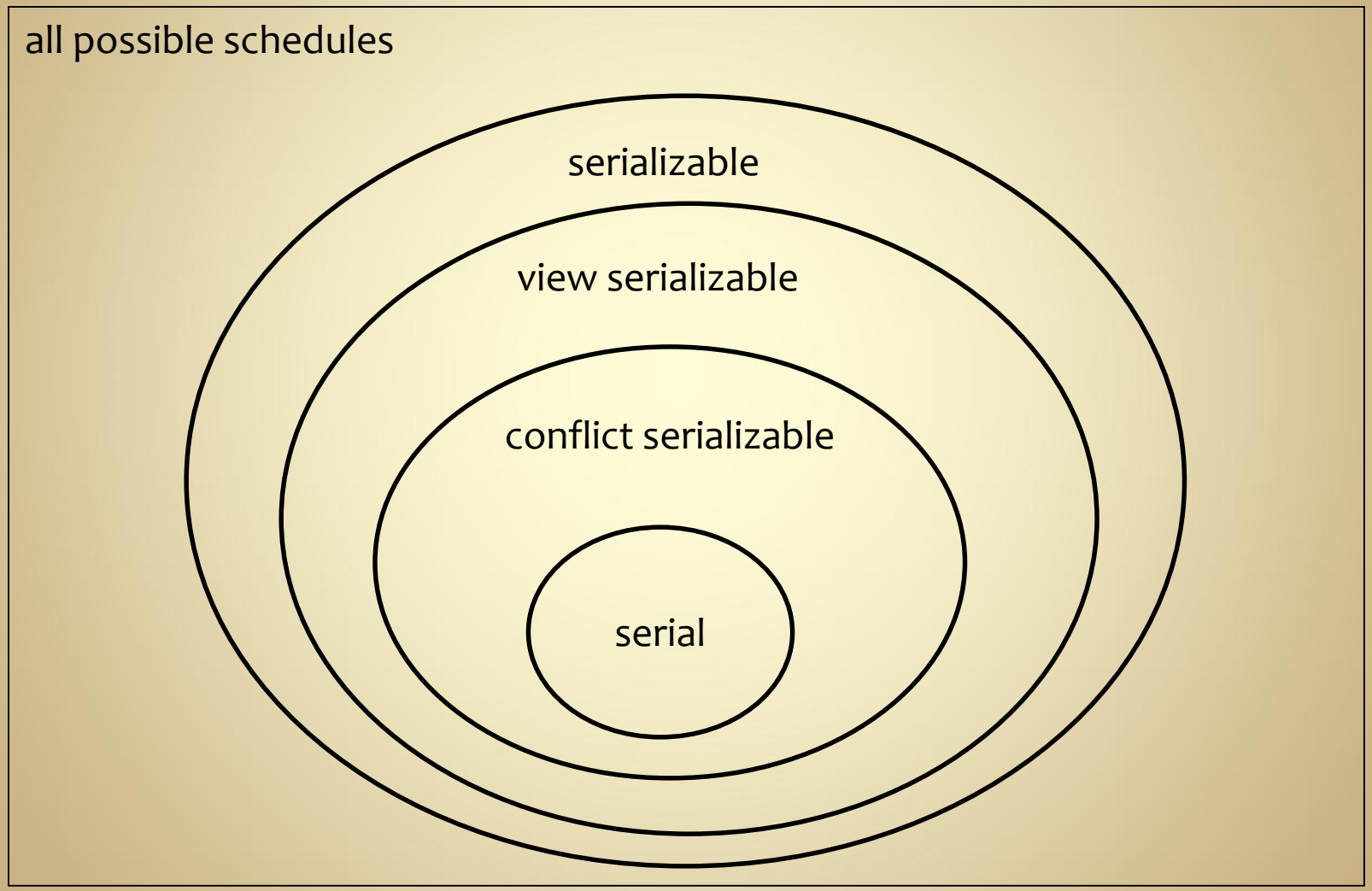
Advanced Database Management Systems

Recoverability Classes - Chapter 17
Concurrency Control – Chapter 18

Serializability Classes



Serializability

- All ***serial*** schedules are correct schedules
- A schedule is ***serializable***
  if it is equivalent to some serial schedule
  - all serializable schedules are correct schedules
- ***view-serializable*** schedules can be shown to be serializable by applying view-equivalence
  - view-equivalence is NP-hard
- ***conflict-serializable*** schedules can be shown to be serializable by applying conflict-equivalence
  - conflict-equivalence is relatively cheap
- Serializablility cannot always be proven
  - some correct schedules are rejected by serializability tests

Serializability Example

Is this schedule serializable?

r1(x) r2(z) r1(z) r3(x) r3(y) w1(x) w3(y) r2(y) w2(z) w2(y)

Ordering of conflicting operations:

a: r1(z) < w2(z)

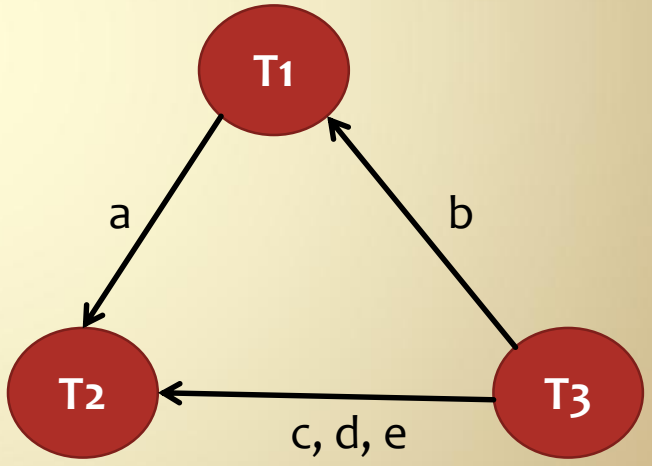
b: r3(x) < w1(x)

c: r3(y) < w2(y)

d: w3(y) < r2(y)

e: w3(y) < w2(y)

Precedence graph:



no loops in graph → schedule is conflict-serializable

→ schedule is correct

Serializability Example

Is this schedule serializable?

r1(x) r2(z) r3(x) r1(z) r2(y) r3(y) w1(x) w2(z) w3(y) w2(y)

Ordering of conflicting operations:

a: r3(x) < w1(x)

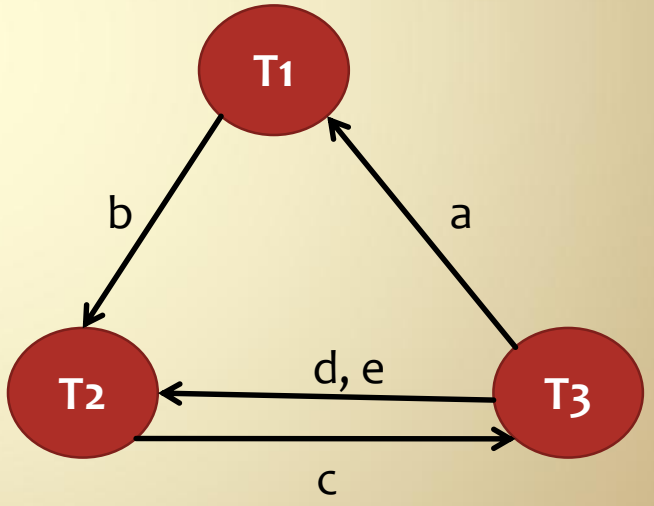
b: r1(z) < w2(z)

c: r2(y) < w3(y)

d: r3(y) < w2(y)

e: w3(y) < w2(y)

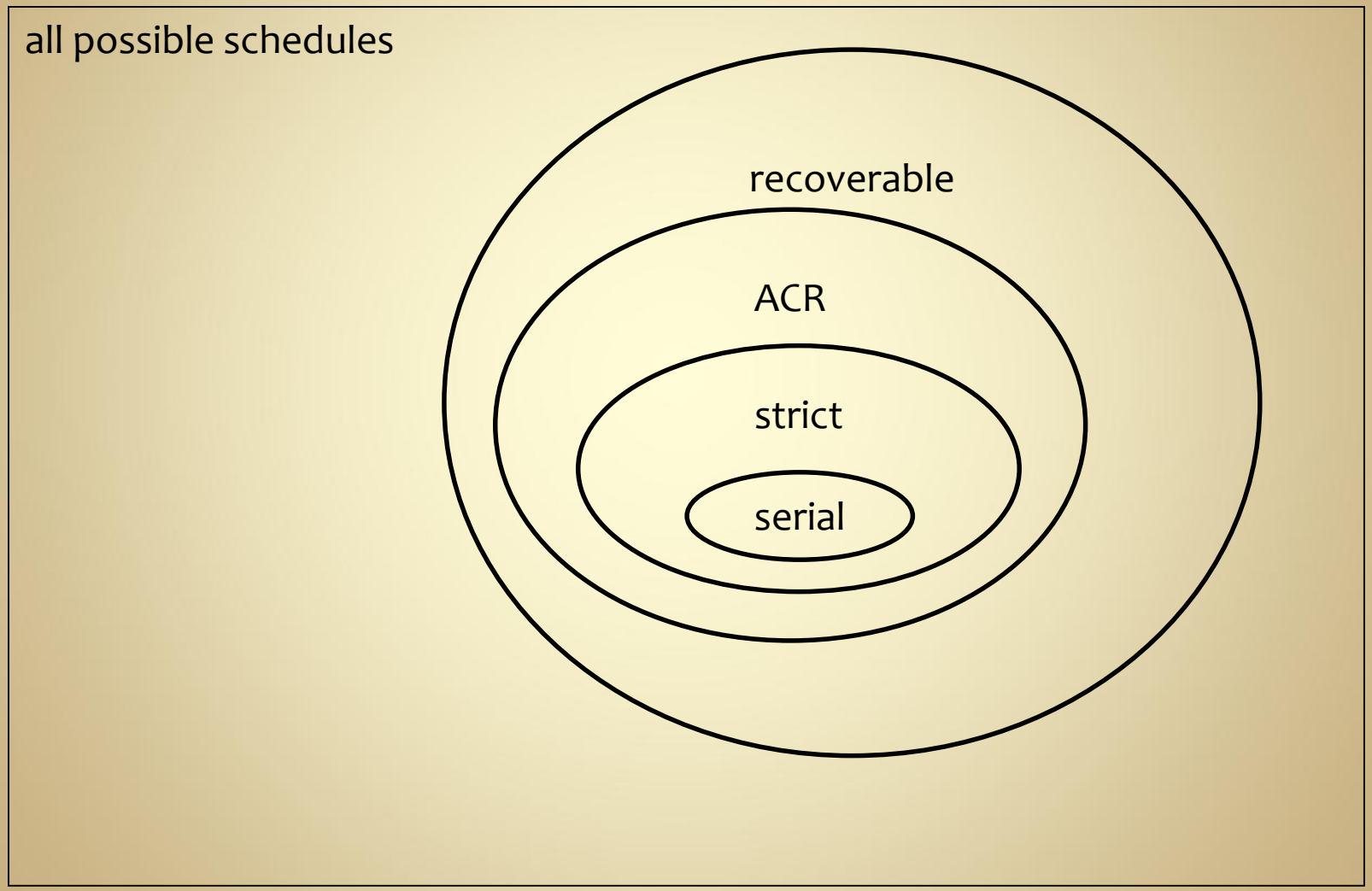
Precedence graph:



loops in graph → schedule is not conflict-serializable

→ schedule is not provably correct

Recoverability Classes



Recoverability Classes

- Recoverability indicates whether a schedule will allow for recovery in the case of a transaction failure
- If a schedule is ***recoverable***, it will never be necessary to roll-back a committed transaction
  - any potential problems can be handled by aborting non-committed transactions
- Other recoverability classes indicate the ease of recovery for schedules in that class

- Recoverability is not an indicator of correctness

Recoverable Schedules

- In a *recoverable schedule*, a committed transaction never needs to be rolled back.
  - a transaction cannot be committed if it is potentially involved in an incorrect schedule
- Recoverable schedule test:
  - ***no transaction T commits until all transactions that wrote something that T reads have committed***
  - test prevents T from committing if it uses data that might later become invalid

ACR Schedules

- ACR = Avoids Cascading Rollbacks

- Cascading Roll-back:
  If an uncommitted transaction T1 reads data written by transaction T2, and T2 is rolled-back, then T1 also has to be rolled-back
  - the roll-back cascades from T2 to T1

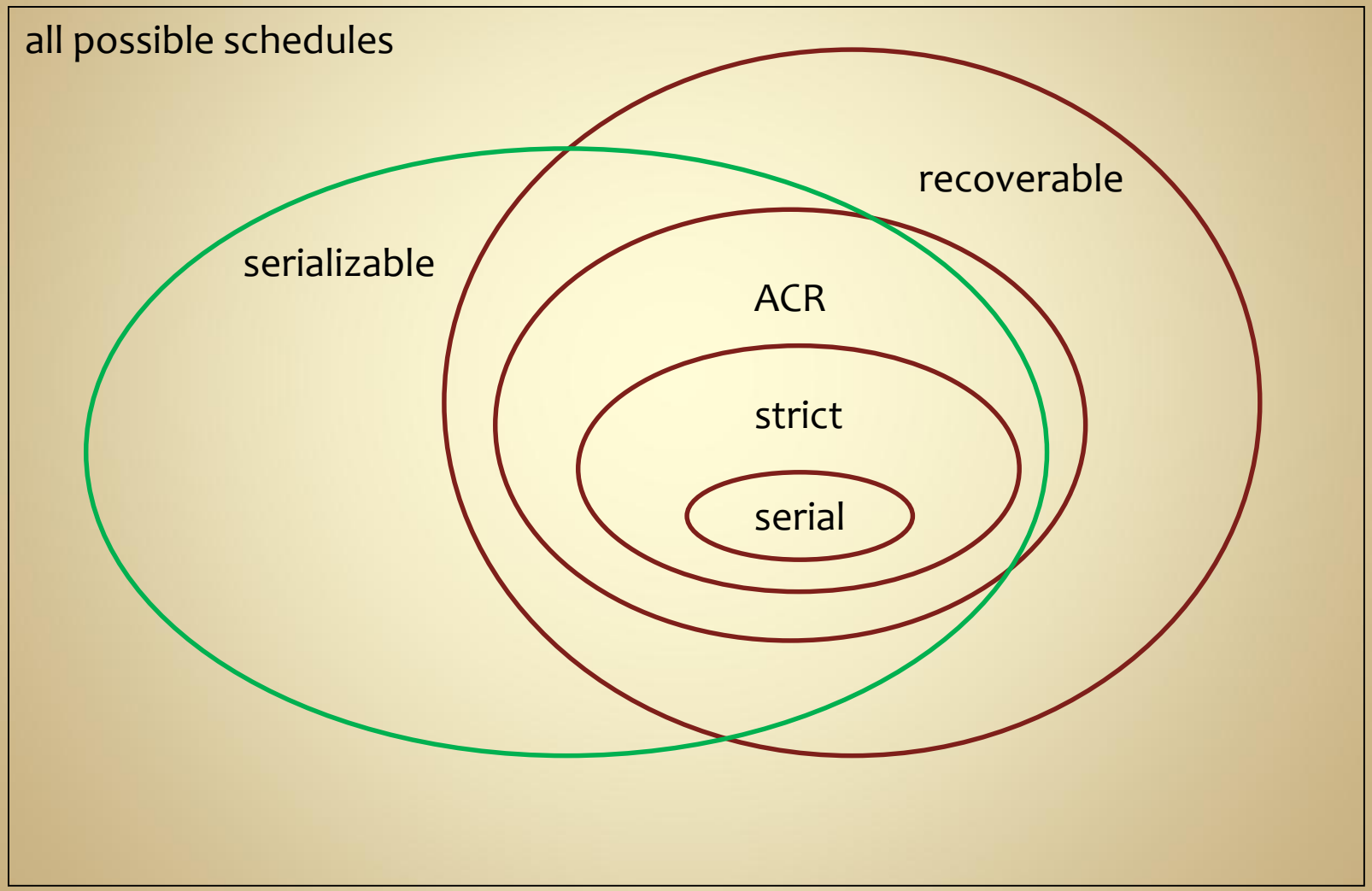
- ACR test:
  - ***every transaction only reads things written by committed transactions***

Strict Schedules

- In a *strict schedule*, any transaction can be aborted by simply restoring the values of any object that it wrote

- Strict schedule test:
  - ***no transaction can read or write anything that was written by an uncompleted transaction***

Classes of Schedules



Concurrency Control

- Concurrency control is the enforcement of policies regarding allowed schedules
- Minimal policy:
  - never allow a schedule that is not in (serializable U recoverable)
- Other possible policies:
  - allow only serial schedules (no concurrency)
  - allow only serializable, ACR schedules
  - allow only strict schedules

SQL: CC and Transactions

- **`SET TRANSACTION`**
  sets the transaction *access mode*:
  - READ ONLY → only allows SELECT
  - READ WRITE → allows SELECT, UPDATE, INSERT, DELETE, CREATE

- **`SET TRANSACTION ISOLATION LEVEL`**
  sets the transaction *isolation level*:
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

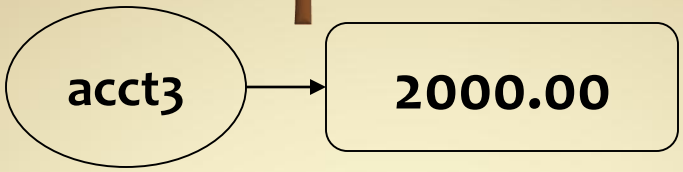
SQL Transaction Violations

- **Dirty reads:** occur when a transaction reads a record altered by another transaction that has not yet completed
- **Non-repeatable reads:** occur when one transaction reads a record while another transaction modifies it
- **Phantom records:** occur when a transaction reads a group of records, then another transaction changes the set of records that would have been read

SQL Isolation Levels

isolation level	prevents
READ UNCOMMITTED	
READ COMMITTED	dirty reads
REPEATABLE READ	dirty reads non-repeatable reads
SERIALIZABLE	dirty reads non-repeatable reads phantom records

The Lost Update Problem



XACT 1

XACT 2

reads 2000

r2(acct3)

reads 2000

r1(acct3)

computes 1700

XACT1: acct3 := acct3 – 300

XACT2: acct3 := acct3 + 900

computes 2900

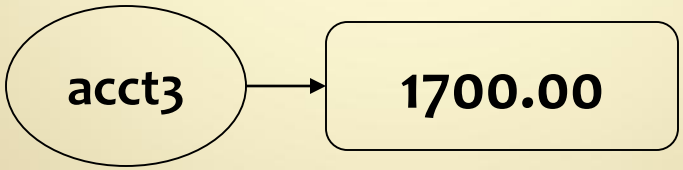
writes 2900

w2(acct3)

writes 1700

w1(acct3)

this update was lost



Lost Update: Changes made by one transaction are overwritten by changes from another transaction.

Dirty Read

Schedule: b1, r1(X), w1(X), b2, r2(X) …

T2 has read the value written by T1.
What if T1 aborts?

Dirty Read: Accessing data that is not yet committed.
Dirty reads can cause cascading roll-backs.

Non-repeatable Read

Schedule: b1, r1(X), b2, r2(X), w1(X), …

T1 has changed the value read by T2.
The value held by T2 is no longer valid
(or valid only if T1 aborts).

Non-repeatable Read: Data was changed since it was read.
If data is read again, a different value will be seen.

Phantom Records

- T1: select accountNum from Account
  where balance > '$1000.00‘
- T2: update Account
  set balance = balance –'$500.00'
  where accountNum = 387
- If T1 reruns its query, the record for account 387 might no longer be included.

Phantom Records: The set of records read by a transaction is changed by another transaction.

This can also cause problems with aggregate queries.

Concurrency Control

- Transaction theory classifies possible schedules in terms of recoverability and correctness.
  - In theory, we can talk about modifying the schedules to gain desired characteristics
  - In practice, the schedule is determined by the real-time order in which the operations arrive and the only "rescheduling" that is possible is delaying certain operations
- Concurrency control implements mechanisms to achieve specific policies

- general protocol classifications:

  - pessimistic: prevent unwanted schedules from occurring even if it reduces concurrency and stalls transactions

  - optimistic: allow any schedule, later abort transaction(s) contributing to unwanted schedules

CC protocols

- CC protocols are the specifications of the mechanisms used to achieve specific policies
- general protocol classifications:
  - optimistic: allow any schedule, later abort transaction(s) contributing to unwanted schedules

Pessimistic CC Protocols

- ***Pessimistic***: prevent unwanted schedules from occurring even if it reduces concurrency and stalls transactions
- ***Locking protocols***:
  Delay conflicting operations by *locking* data items
  - locking is most common CC protocol
  - reduces concurrency
  - may result in deadlock, livelock or starvation
- ***Timestamping protocols***:
  Abort transactions that request operations that violate serializability
  - timestamps used to order conflicting operations

Pessimistic CC Protocols

- Locking
  - locking is most common CC protocol
  - reduces concurrency
  - may result in deadlock, livelock or starvation
  - 2PL allows only serializable schedules
  - 2PL never requires xact roll-backs due to conflict
- Time-stamping
  - allows only serializable schedules
  - cannot result in deadlock
  - may cause cascading aborts due to conflict
  - may cause starvation

2PL = two phase locking … defined below

Optimistic Protocols

- no checking is done while a transaction is executing
  - optimistically assume everything will be fine
- all operations are performed on local copies of data items
- validity is checked when transaction commits
  - invalid transactions determined at latest possible time
- maximum concurrency
- may cause aborts due to conflict
- no possibility of deadlock

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

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

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 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 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 be 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