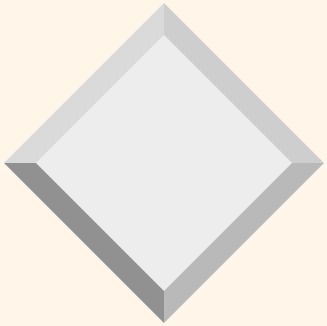
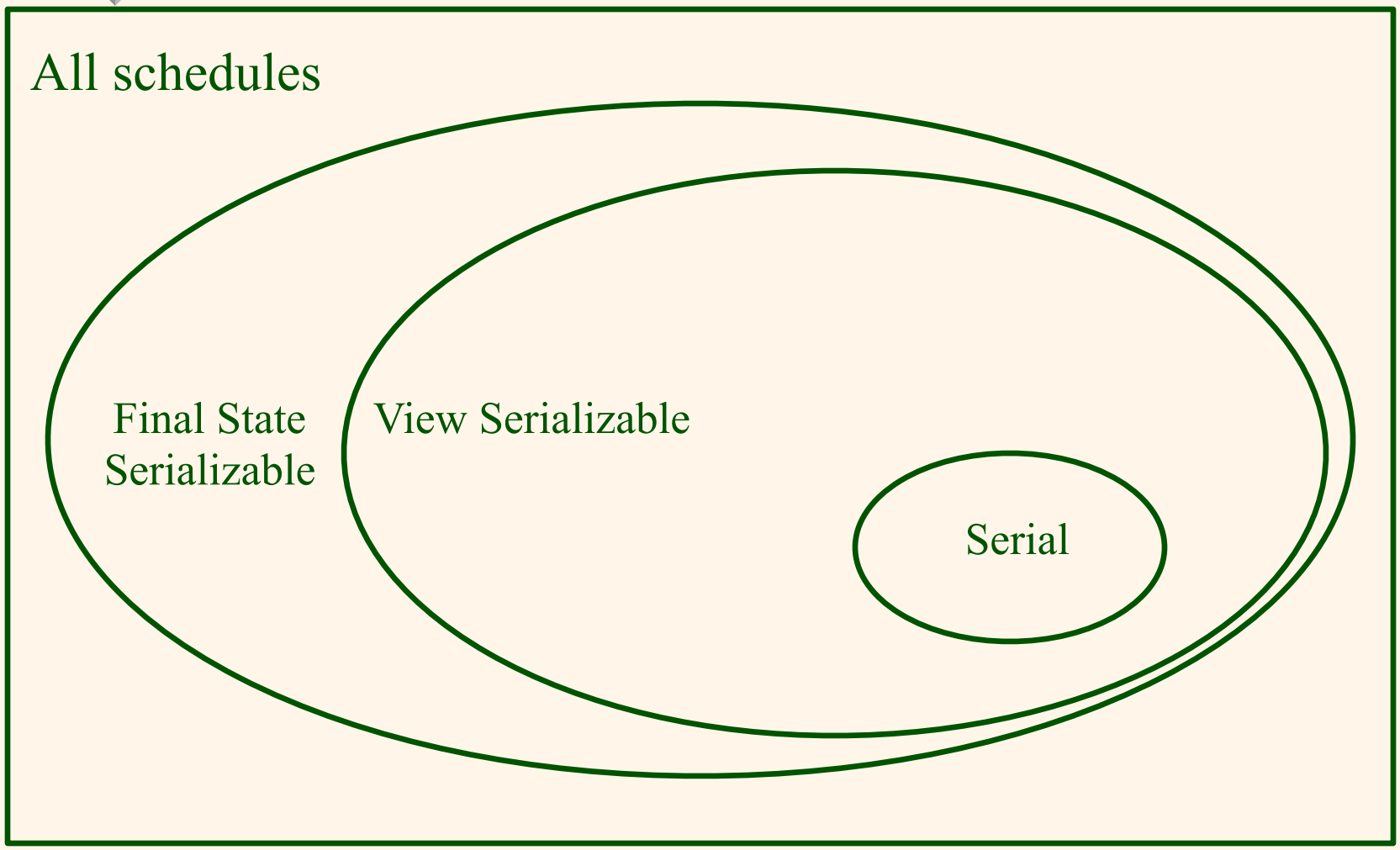


Concurrency Control

Last time

- Isolation anomalies
  - Dirty read, unrepeatable read, lost update
- Towards defining "correct" execution
  - Transaction schedules
    - e.g. R1(A) W2(B) R2(C)
  - "Good" schedules have some equivalence to a serial schedule with same transaction

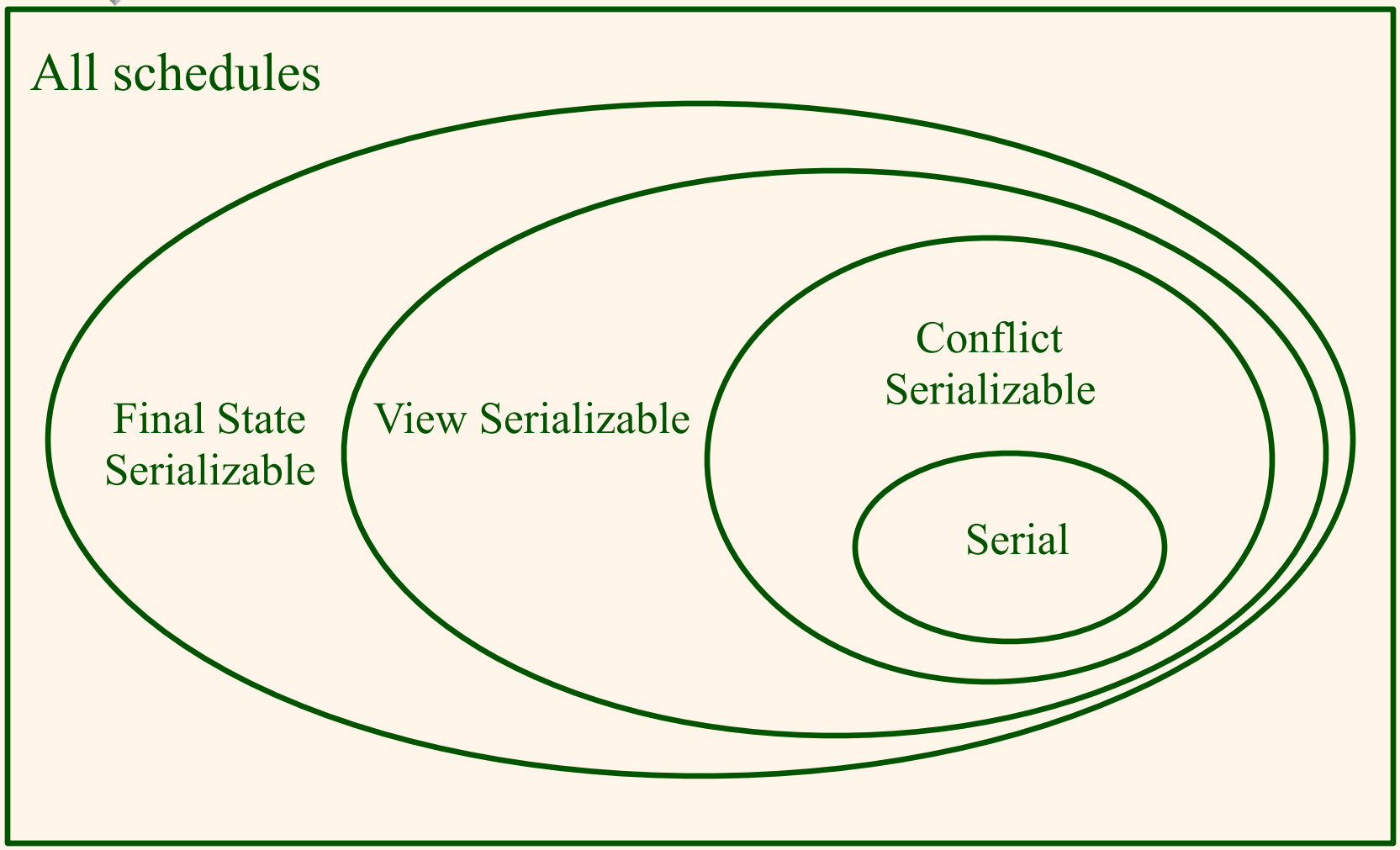
Big Picture (all inclusions are proper)



Another notion of serializability

- View-serializability is nice but difficult to check/enforce in practice
- A third notion: conflict-serializability
- This can be enforced efficiently

Big Picture (all inclusions are proper)



Conflicting Operations

- Based on identifying conflicting pairs of operations between transactions
- Observation: read-only transactions cannot interfere with each other in any bad way!
- The only problems are due to writes
  - All three of our anomalies involve a write somewhere

Conflicting Operations

- Two operations by different transactions on the same object conflict if at least one is a write
  - WW
  - WR
  - RW
- Example: if T1 and T2 both write Alice's bank account balance, they conflict
- But if T1 only writes Alice's balance and T2 only writes Bob's, no conflict!
- Also, two writes by the same transaction never conflict with each other

Conflicting Operations

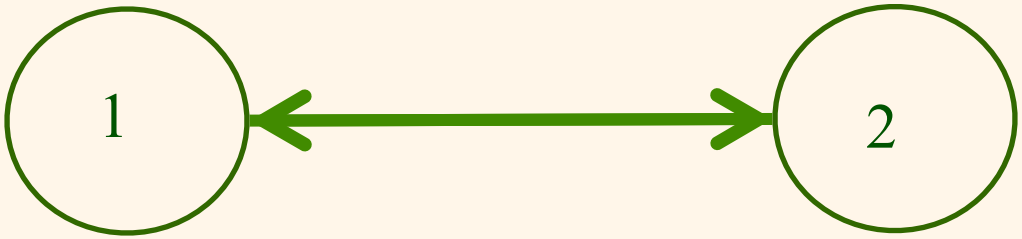
- Conflicting operations do not commute
  - W1(A) W2(A)
  - R1(A) W2(A)
  - W1(A) R2(A)
- So, they induce some notion of precedence or ordering

Conflict Graphs

- Given a schedule, can identify all conflicting pairs of operations and represent them as a graph
- Nodes are transactions
- Edge from i to j if transaction i contains an operation that conflicts with and precedes (in the schedule) an operation by transaction j
- Example: R1(A) W2(A) R1(A)

Conflict Graphs

- R1(A) W2(A) R1(A)

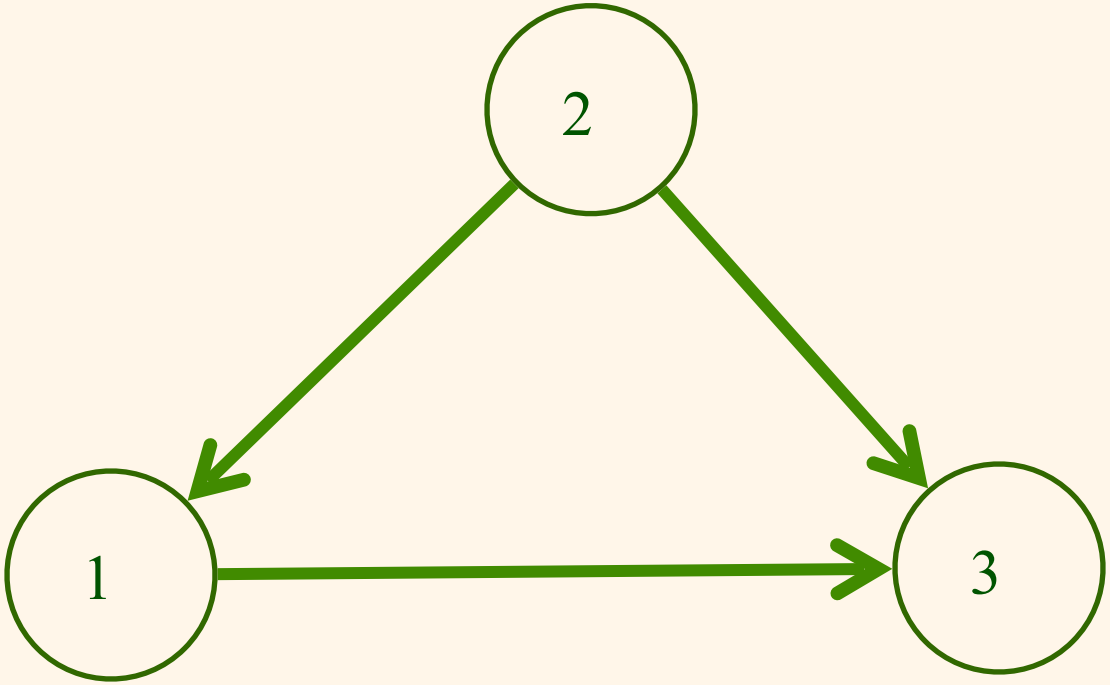


A conflict graph exercise

R1(A) R2(A) R1(C) W1(A) R3(C) W2(B) W3(B) W3(C)

A conflict graph exercise

R1(A) R2(A) R1(C) W1(A) R3(C) W2(B) W3(B) W3(C)

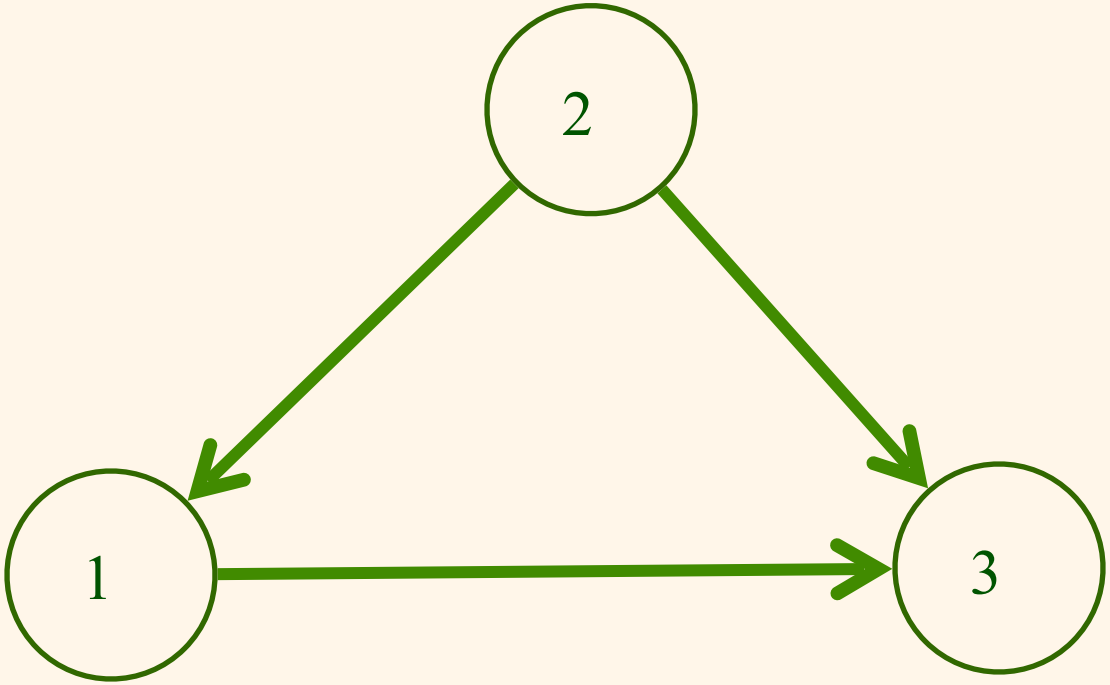


Conflict Serializability

- A schedule is conflict serializable if its conflict graph contains no cycle
- Alternative (equivalent) statement: it is conflict serializable if it has the same conflict graph as some serial schedule
  - Why are these equivalent?
- Topological sort on the conflict graph gives us equivalent serial execution

A conflict graph exercise

R1(A) R2(A) R1(C) W1(A) R3(C) W2(B) W3(B) W3(C)



An equivalent serial schedule

R1(A) R2(A) R1(C) W1(A) R3(C) W2(B) W3(B) W3(C)

- Executing the transactions in the order 2, 1, 3 produces the same DB
- Let's see if we believe it…

R2(A) W2(B) R1(A) R1(C) W1(A) R3(C) W3(B) W3(C)

A slightly different schedule

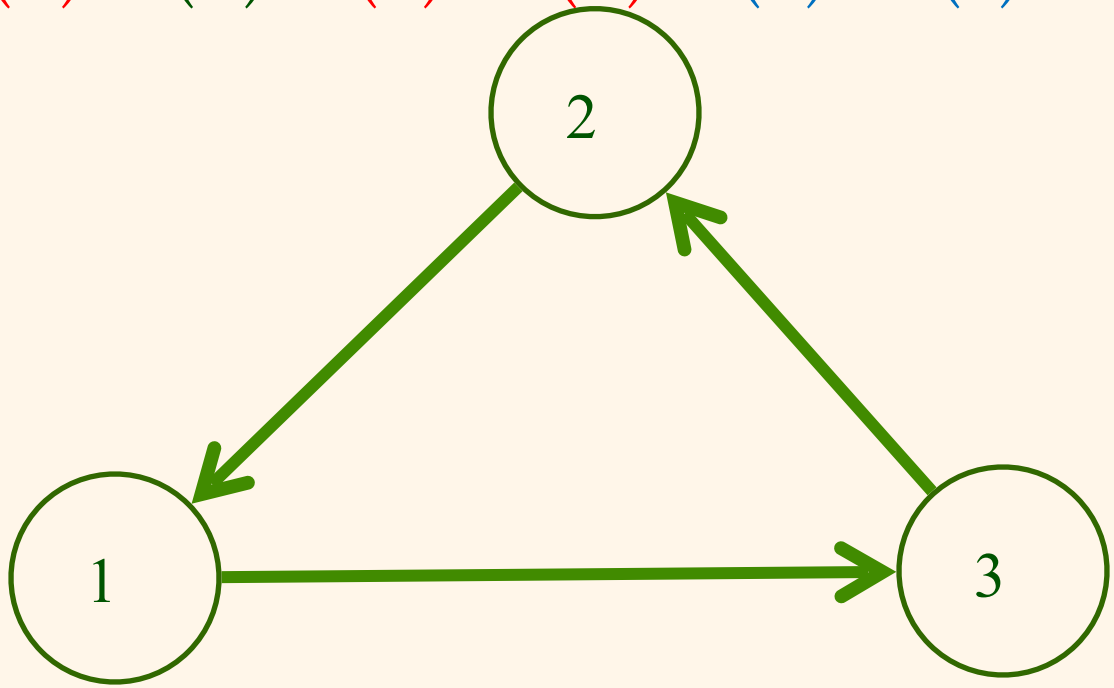
R1(A) R2(A) R1(C) W1(A) R3(C) W3(B) W2(B) W3(C)

Compare with previous:

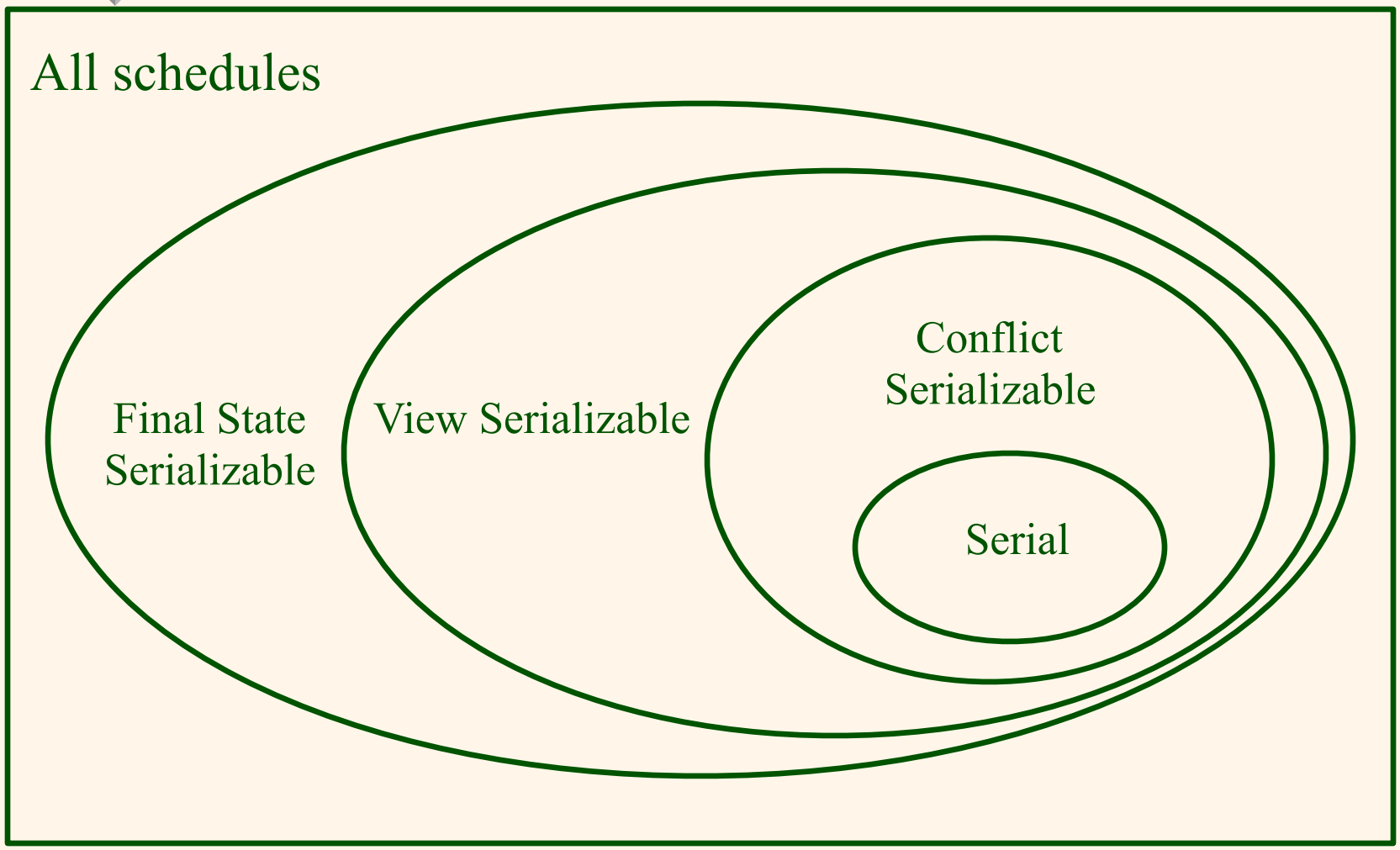
R1(A) R2(A) R1(C) W1(A) R3(C) W2(B) W3(B) W3(C)

A conflict graph exercise

R1(A) R2(A) R1(C) W1(A) R3(C) W3(B) W2(B) W3(C)



Big Picture (all inclusions are proper)



What about aborts?

- Convention: if a schedule contains transactions that abort, we ignore the operations (R/W) of these transactions when determining if a schedule is serializable
- Rationale: aborted transactions' operations should not have happened and will be cleaned up
- Although, the "cleanup" can be nontrivial!

Dealing with aborts

- Cleanup issues
  - W1(A) R2(A) W2(B) C2 A1 ... <= not good!
  - 2 has read a value of A that "should not have been there"
  - 2 has already committed, so extremely unclear how to clean up in a sensible way

Recoverability

- A schedule is recoverable if a transaction commits only after the commit of all transactions whose changes it has read
  - Never have to undo an *already committed* transaction due to someone else's abort
  - W1(A) R2(A) W2(B) C2 ... <= not recoverable

Recoverability

- Enforcing this will be a problem - need to keep track of who has read from whom
  - So, more (stronger) criteria

ACA schedules

- A schedule avoids cascading aborts if a transaction never reads uncommitted writes
  - Now that is easier to enforce
  - W1(A) R2(A) W2(B) C1 C2 is recoverable but does not avoid cascading aborts
- But there could still be problems even if we enforce ACA

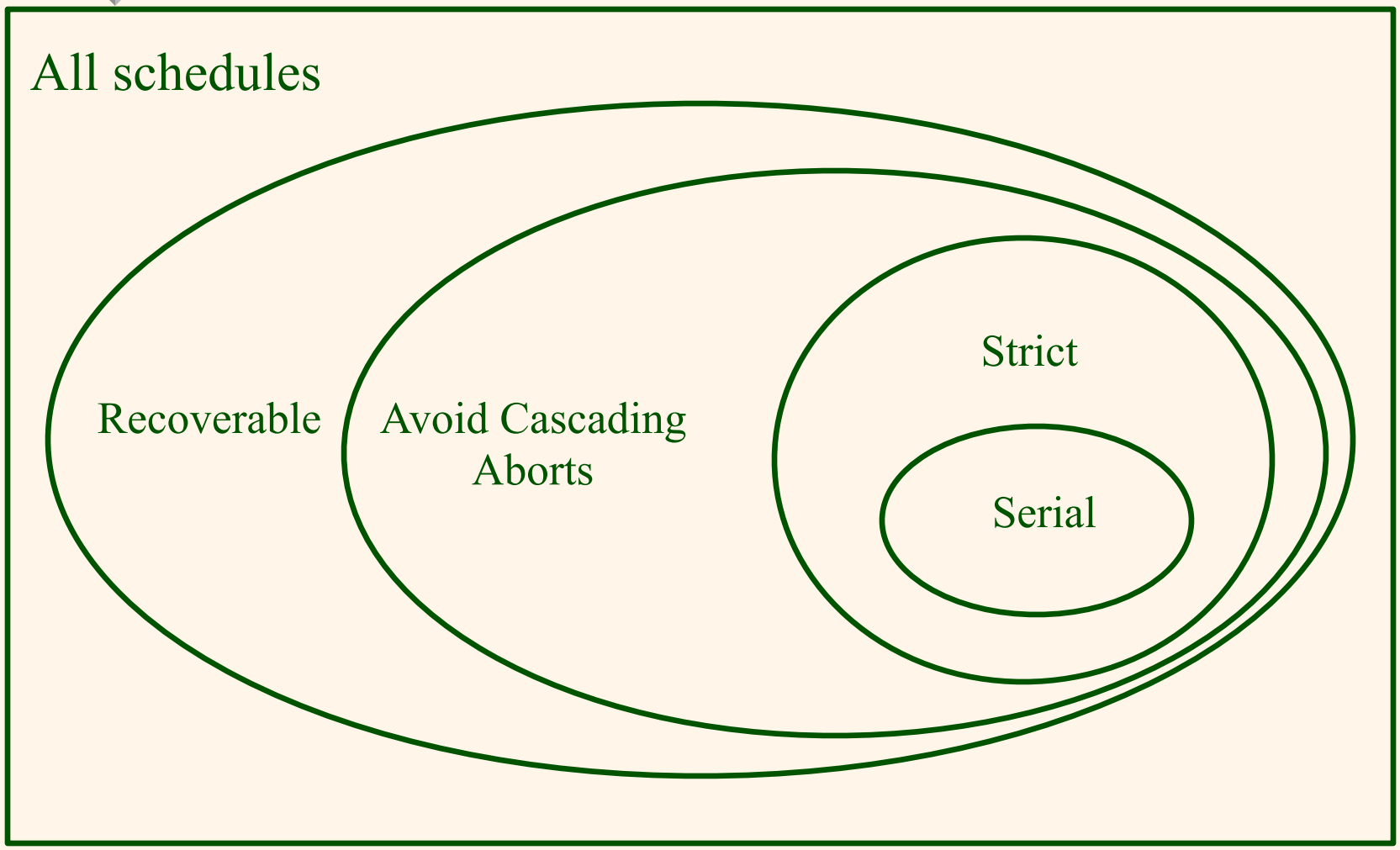
Strict schedules

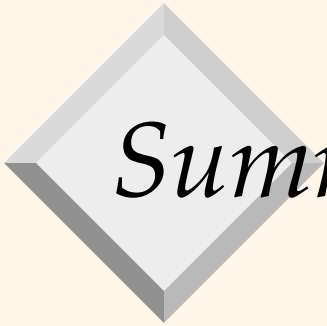
- Consider schedule W1(A) W2(A) C2 A1
- Recoverable and avoids cascading aborts because no-one never read anything
- But "cleanup" could be a pain
  - Need to retain write by T2 while erasing write by T1
  - Depending on what the objects are and what records you keep of changes, could be a lot of work
  - May need to roll back to initial state and redo T2

Strict schedule

- If we don't want this problem, can place an even stronger restriction
- A schedule is strict if whenever a transaction writes to a data item, no other transaction can read or write to that item until T has committed or aborted
- Avoids our problem W1(A) W2(A) C2 A1

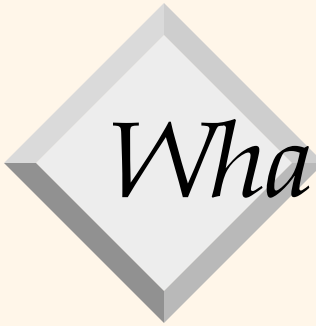
Summary


All schedules
Recoverable
Avoid Cascading Aborts
Strict
Serial



Summary

- Restrictions on schedules for serializability:
  - Final-state serializable
  - View-serializable
  - Conflict-serializable
- Restrictions on schedules for handling aborts:
  - Recoverable
  - Avoids Cascading Aborts
  - Strict



What was all this for?

- We now know how to tell good interleavings from bad ones
  - Want either view-serializability or conflict-serializability
  - And possibly something like ACA or strictness too
- The system needs to allow only good interleavings
- Let's see this in action in MySQL

Transaction 1

```
START TRANSACTION;

UPDATE accounts
SET amount= amount-100
WHERE name = 'Alice';

UPDATE accounts
SET amount= amount+100
WHERE name = 'Bob';

COMMIT;
```

Transaction 2

```
START TRANSACTION;

UPDATE accounts
SET amount= amount*1.1
WHERE name = 'Alice';

UPDATE accounts
SET amount= amount*1.1
WHERE name = 'Bob';

COMMIT;
```

Bad interleaving

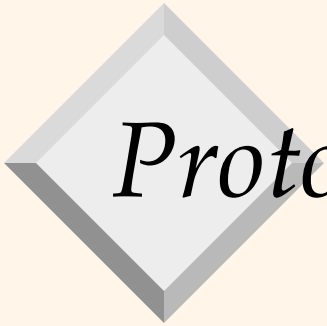
Transaction 1	Transaction 2
UPDATE accounts SET amount= amount- 100 WHERE name = 'Alice';	
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Alice';
	UPDATE accounts SET amount= amount*1.1 WHERE name = 'Bob';
UPDATE accounts SET amount= amount+ 100 WHERE name = 'Bob';	

Enforcing conflict serializability

- Goal: system must only allow interleavings (schedules) that are (conflict) serializable.
- Still don't know how to do it
  - Specification vs implementation
  - There may be (and are!) different ways to do it

Protocols for enforcing a Property

- General systems discussion
- Common problem: need to enforce a property
  - All interleavings of transactions are conflict-serializable
  - No customer receives a product without having paid for it
  - If someone modifies your 4320 grade, that person is one of the 4320 course staff
- To enforce it, system uses an algorithm or protocol
  - E.g. require me to log in before I modify your grade



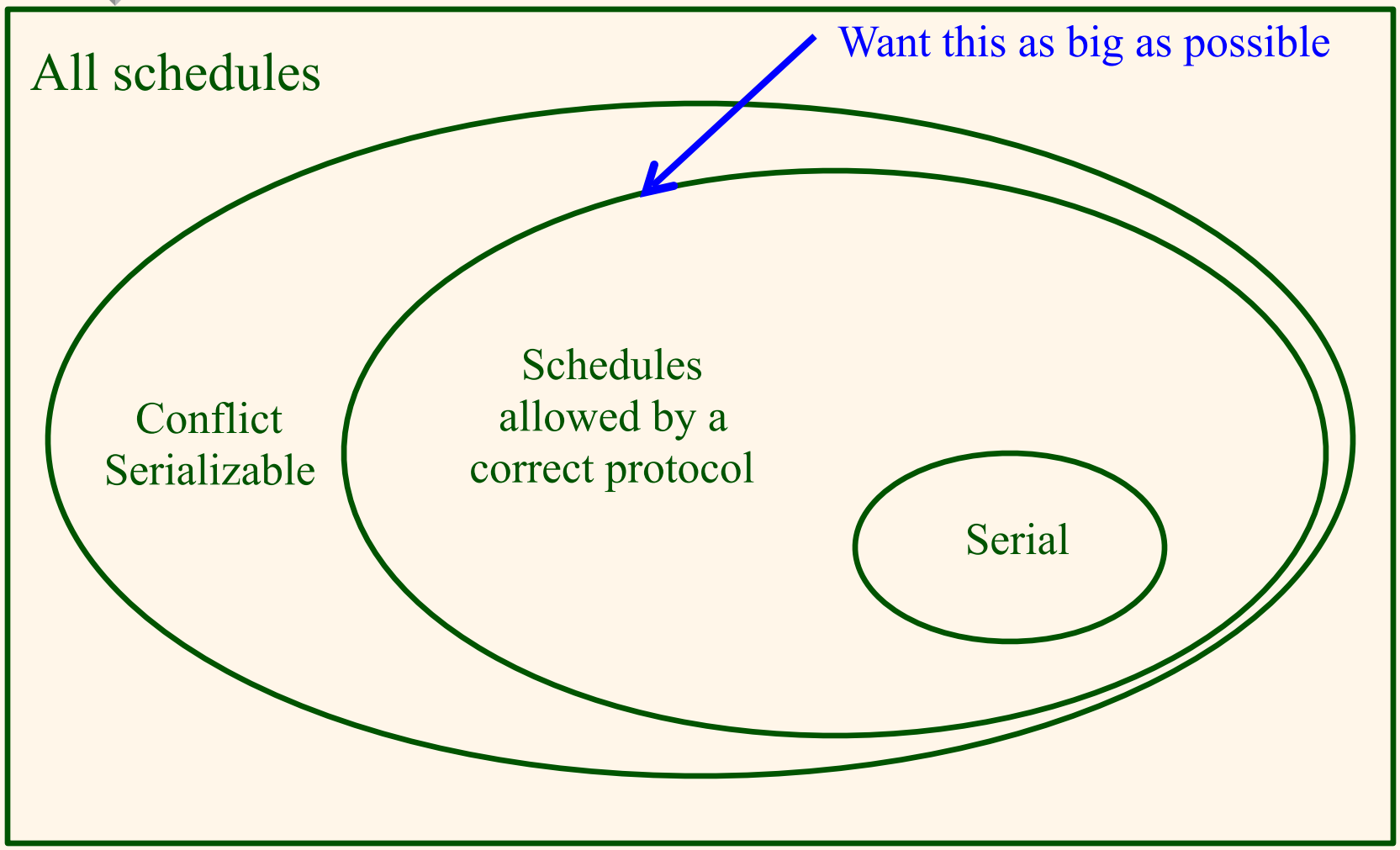
Protocols and properties

- There can be many different protocols for enforcing a property
- Some protocols can be stricter than necessary, but still correct
  - I.e. forbid all "bad" behavior
  - But forbid some "good" behavior too
  - Ok as long as never allow "bad" behavior
  - This often happens in practice

Back to transactions

- Property: conflict-serializability
- A variety of protocols
- Most of which are conservative (disallow some conflict serializable schedules)

Big Picture


All schedules
Want this as big as possible
Conflict Serializable
Schedules allowed by a correct protocol
Serial

Concurrency control protocols

- A very simple protocol: force all transactions to execute serially
  - No operations by different transactions may interleave
  - If another transaction is not done reading/writing to DB, you must wait
- Does this permit only conflict-serializable schedules?
- Does this permit all conflict-serializable schedules?

Reasonable protocols

- Should allow as many conflict-serializable schedules as possible
- Should be simple and not impose too much overhead of their own
  - Otherwise the performance boost from allowing interleavings is lost
- Several (families of) solutions