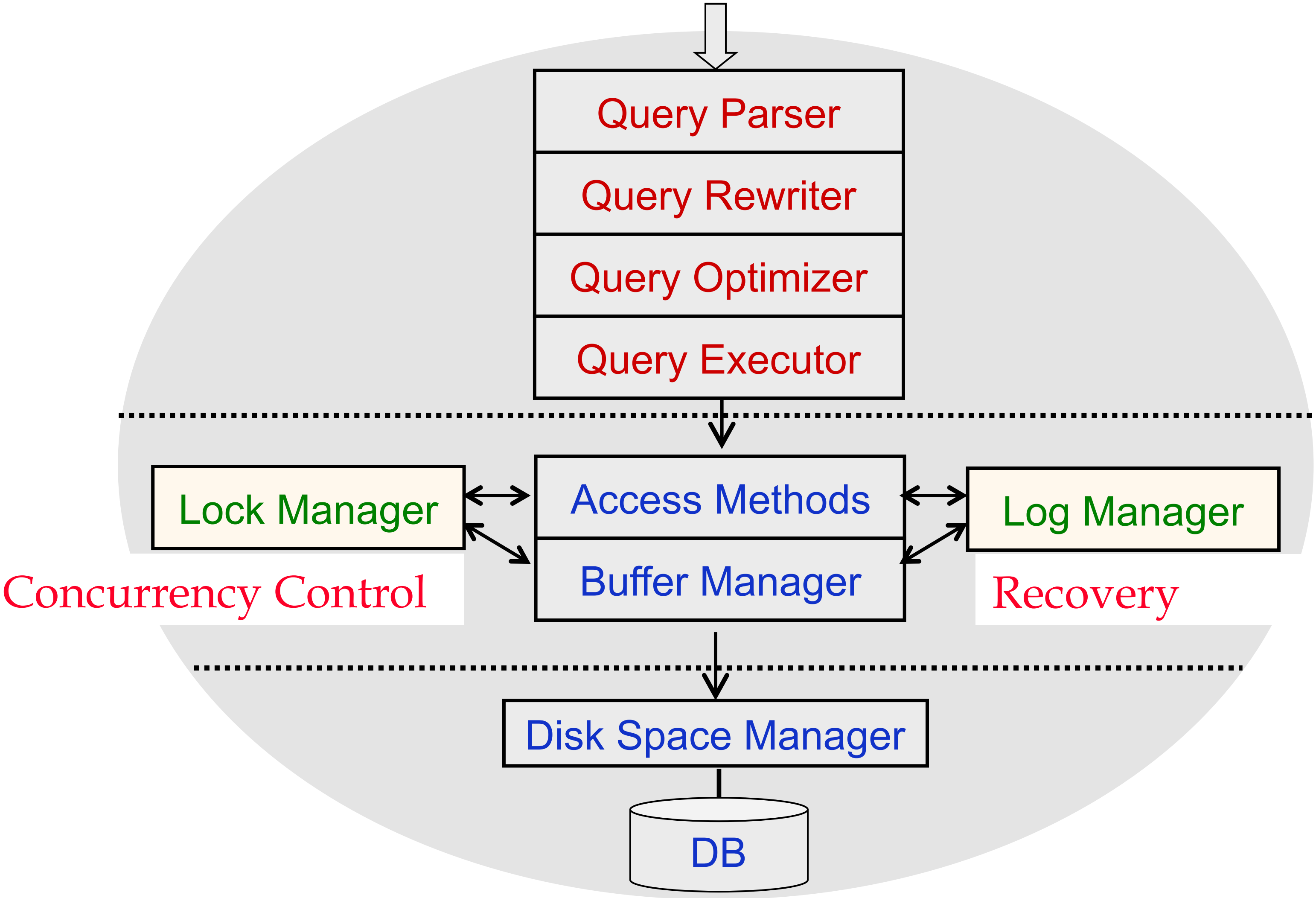
Transaction Management: Concurrency Control

Yanlei Diao

Slides Courtesy of R. Ramakrishnan and J. Gehrke

DBMS Architecture



Outline

- ❖ Transaction management overview
- ❖ Serializability & recoverability
- ❖ Lock-based concurrency control
- ❖ Optimistic concurrency control
- ❖ Efficient B+tree locking

Transactions

- User programs may do many things on the data retrieved.
  - E.g., operations on Bob's bank account.
  - E.g. transfer of money from account A to account B.
  - E.g., search for a ticket, think about it…, and buy it.
- But the DBMS is only concerned about what data is read from/written to the database.
- A *transaction* is DBMS's abstract view of a user program, simply, *a sequence of reads and writes*.

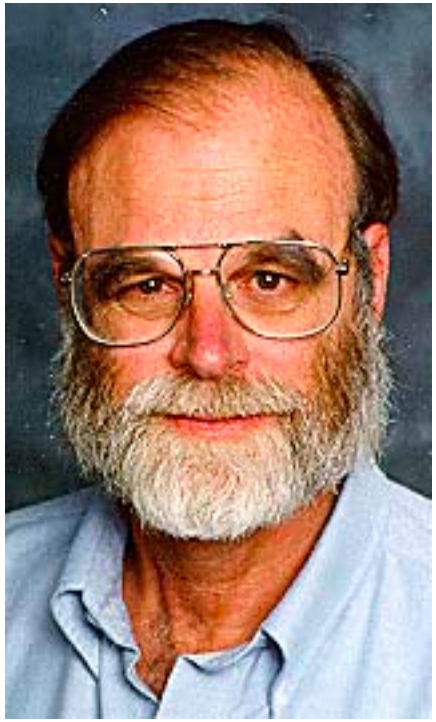
Concurrency

- Many users submit xacts, but each user thinks of his as executing by itself.
  - DMBS *interleaves* reads and writes of xacts for concurrency.
- *Consistency*: each xact starts and ends with a consistent state (i.e., satisfying all *integrity constraints*).
  - E.g., if an IC states that all accounts must have a positive balance, no transaction can violate this rule.
- *Isolation*: execution of one xact appears isolated from others.
  - Nobody else can see the data in its *intermediate state*, e.g., account A being debited but B not being credited.

Recovery

- A transaction might *commit* after completing all its actions, or it could be *aborted* after executing some actions.
- *Atomicity*: either all actions of a xact are performed or none of them is (*all-or-none*).
  - DBMS *logs* all actions so that it can *undo* the actions of aborted xacts.
- *Durability*: once a user program has been notified of success, its effect will persist despite system failure.
  - DBMS *logs* all actions so that it can *redo* the actions of committed xacts.

James Gray & Turing Award



- Jim Gray won Turing Award in 1998 for

  "for seminal contributions to database and transaction processing research and technical leadership in system implementation"

Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Optimistic concurrency control
- Efficient B+tree locking

Example

- Consider two transactions:

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

  - 1st xact transfers $100 from B's account to A's.
  - 2nd xact credits both accounts with a 6% interest payment.
  - No guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

- However, the net effect must be *equivalent to* these two transactions running *serially* in some order!

Example (Contd.)

- Consider a possible interleaving *schedule*:

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

- This is OK. But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

The DBMS's view of the second schedule:

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.
  - If each transaction preserves consistency, every serializable schedule preserves consistency.

Serializability

- *Serializability theory* concerns the schedules of transactions that are not (explicitly) aborted.
- Given a set of such xacts, ideally want to allow *any serializable schedule*.
  - Recognizing any serializable schedule is highly complex, if possible.
- Instead, allow only a *subset* of serializable schedules that are easy to detect.

Conflict Serializability

- Two schedules are *conflict equivalent* if:
  - Involve the same actions of the same transactions.
  - Every pair of *conflicting actions* is ordered the same way.
- Schedule S is *conflict serializable* if S is conflict equivalent to some serial schedule.
- Given a set of xacts, conflict serializable schedules are a *subset* of serializable schedules.
  - There are serializable schedules that can't be detected using conflict serializability.

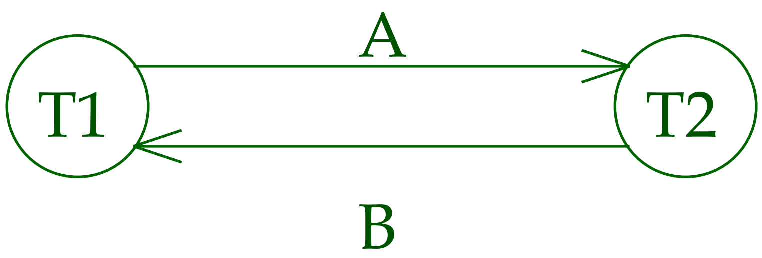
Dependency Graph

- Precedence graph:
  - One node per Xact;
  - Edge from Xact *Ti* to Xact *Tj* if an action of *Ti* *precedes* and *conflicts with* one of *Tj* ‘s actions (*RW, WR, WW operations on the same object*).

- **Theorem**: Schedule is conflict serializable *if and only if* its precedence graph is acyclic.

Example

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	



Precedence graph

- The schedule is not conflict serializable:
  - The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.

Recoverability

- *Recoverability theory* concerns schedules that involve *aborted* transactions.

T1:	R(A),W(A)		Abort
T2:		R(A),W(A) Commit	

Unrecoverable!

- A schedule S is *recoverable* if each xact *commits* only after all xacts from which it read have committed.

Recoverability (Contd.)

T1: R(A),W(A)		Abort		
T2:	R(A),W(A)			Abort

Recoverable, but with cascading aborts.

- S *avoids cascading rollback* if each xact may *read* only those values written by committed xacts.

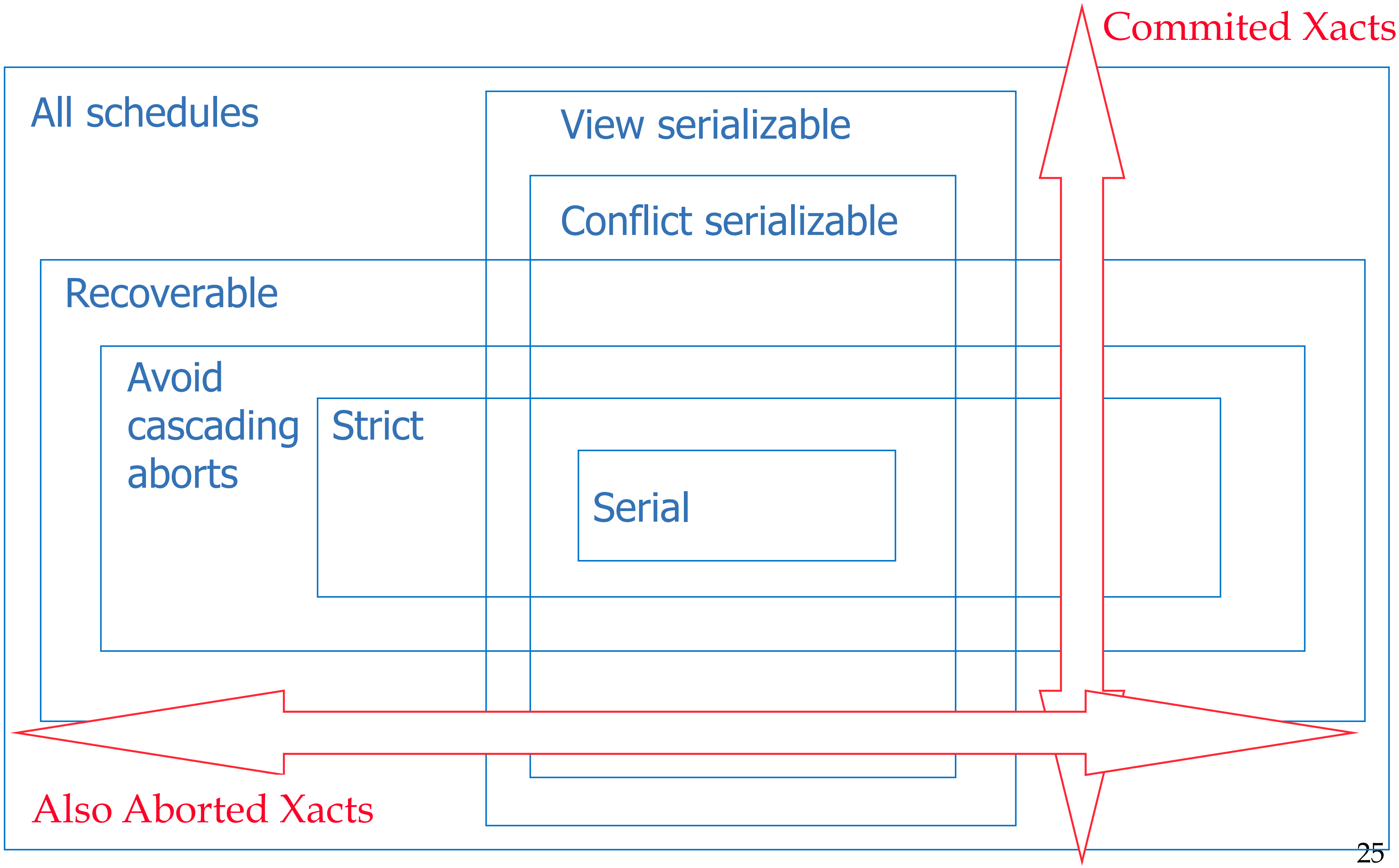
Recoverability (Contd.)

T1:	R(A), W(A)		Abort	
T2: R(A)		W(A)		(Commit)

Recoverable, no cascading aborts,
but update of A by T2 will be lost!

- S is ***strict*** if each xact may *read and write* only objects *previously* written by committed xacts.
  - No cascading aborts.
  - Actions of aborted xacts can be simply undone by restoring the original values of modified objects.

Venn Diagram for Schedules



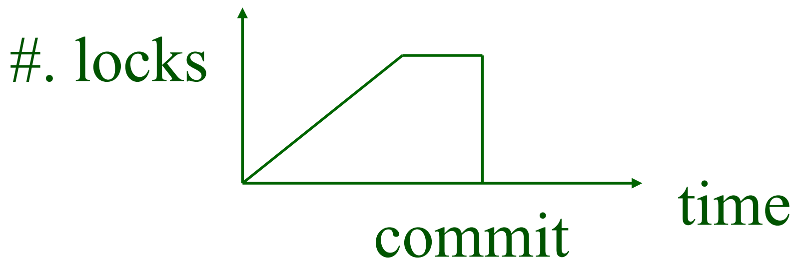
Outline

- Transaction management overview
- Serializability & recoverability
- Lock-based concurrency control
- Optimistic concurrency control
- Efficient B+tree locking

(1) *Locking Protocol: Strict 2PL*

- Strict Two-Phase Locking (Strict 2PL) Protocol:
  1. Each Xact must obtain a S (*shared*) lock on object before reading, an X (*exclusive*) lock on object before writing.
  2. If an Xact holds an X lock on an object, other Xact's cannot get a lock (S or X) on that object and become *blocked*.
  3. All locks held by a xact are released when it completes.
     - Other blocked xact's can resume now.

Compatibility	**Shared**	**Exclusive**
Shared	Y	N
Exclusive	N	N



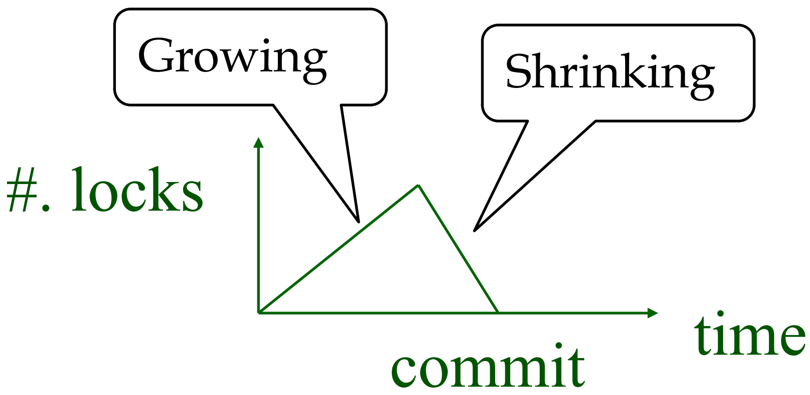
Strict 2PL (contd.)

- **Theorem**: Strict 2PL allows only schedules whose precedence graph is acyclic.
  - Strict 2PL only allows *conflict serializable schedules*!

- Strict 2PL is strict with respect to recoverability.
  - Strict 2PL is *recoverable without anomalies related to aborted transactions*.
  - Hence, it simplifies transaction aborts.

Locking Protocol: Nonstrict 2PL

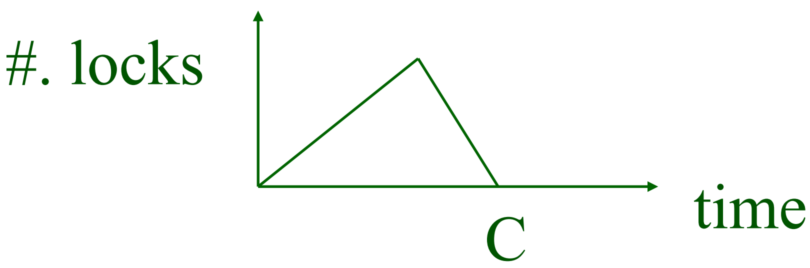
- Nonstrict Two-Phase Locking Protocol
  1. Each Xact must obtain a S (*shared*) lock on object before reading, an X (*exclusive*) lock on object before writing.
  2. If an Xact holds an X lock on an object, other Xact's cannot get a lock (S or X) on that object and become *blocked*.
  3. *A xact cannot request additional locks once it releases any locks.*
     - It releases locks earlier, so blocked xact's can resume earlier.



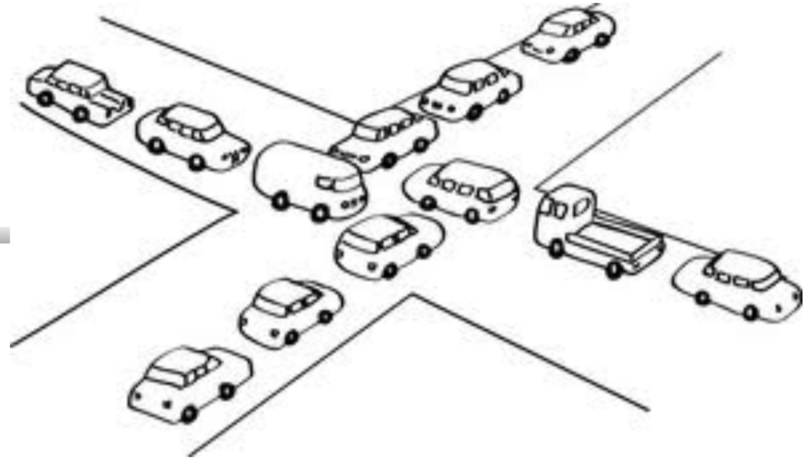
Nonstrict 2PL (contd.)

- **Theorem**: Nonstrict 2PL ensures acyclicity of precedence graph.
  - Nonstrict 2PL only allows *conflict serializable* schedules.
  - An equivalent serial schedule is given by the order of xacts entering their *shrinking phase*.

- Nonstrict 2PL is recoverable but *not strict*!
  - Involves complex abort processing.
  - But allows xacts to go through more quickly.



Deadlocks



- Deadlock: Cycle of transactions waiting for locks to be released by each other.

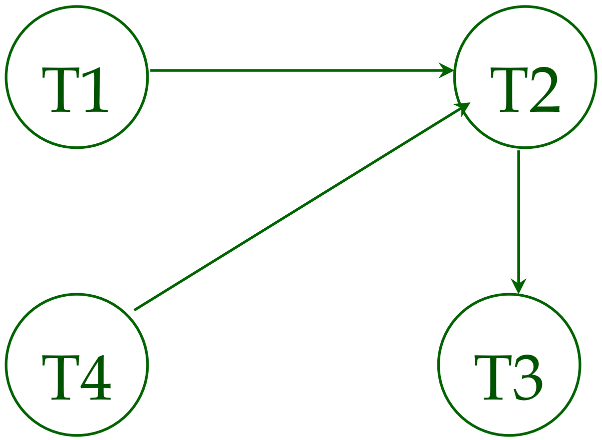
- Two ways of dealing with deadlocks:
  - Deadlock *detection*
  - Deadlock *prevention*

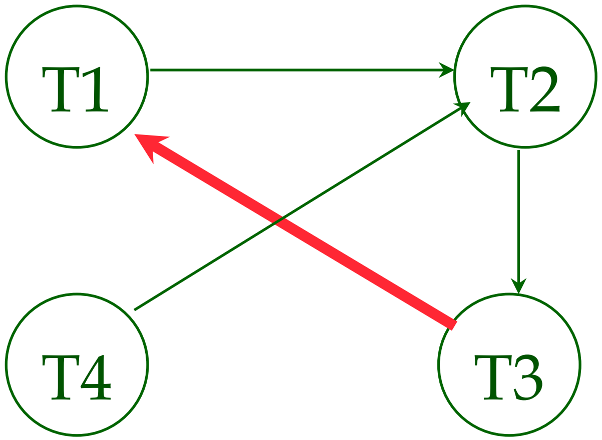
Deadlock Detection

- Create a waits-for graph:
  - Nodes are Xacts.
  - There is an edge from Xact Ti to Xact Tj if Ti is *waiting for* Tj to release a lock.
  - Note the difference from the precedence graph for conflict serializability.
- Periodically check for cycles, indicating deadlocks, in the waits-for graph.
  - Resolve a deadlock by aborting a transaction on the cycle and releasing all its locks.

Deadlock Detection (Contd.)

T1:	S(A), R(A),		S(B)				
T2:		X(B),W(B)			X(C)		
T3:				S(C), R(C)			X(A)
T4:						X(B)	





Deadlock Prevention

- Assign priorities based on timestamps.
  - The older the timestamp, the higher the xact's priority.
- *Wait-Die*: Ti wants a lock that Tj holds. If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts.
  - *Lower priority xacts can never wait.*
- *Wound-wait*: Ti wants a lock that Tj holds. If Ti has higher priority, Tj aborts; otherwise Ti waits.
  - *Higher priority xacts never wait.*
- If a transaction re-starts, make sure it has its original timestamp so its priority increases.