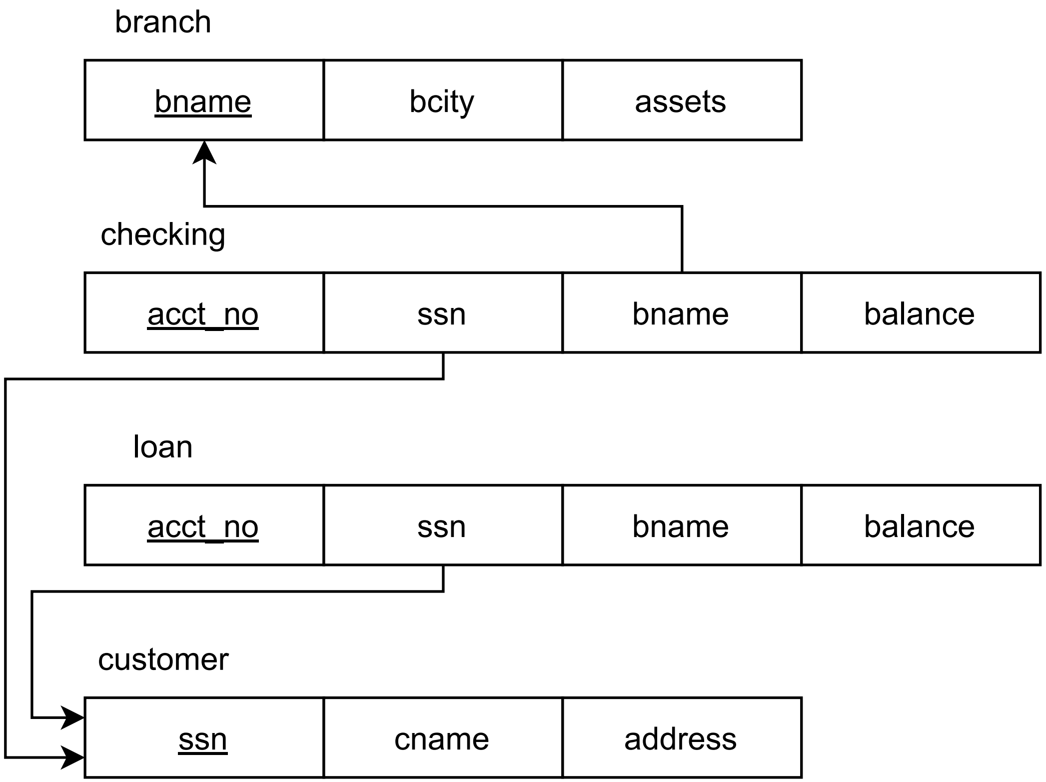
Transactions

CSCI 220: Database Management and Systems Design

Slides adapted from
Simon Miner
Gordon College

Practice Quiz: Query Processing and Optimization

- With a neighbor:
  - Use the heuristics we described in lecture to develop an alternate execution plan for the following query
  - Explain why your plan could be faster



branch

bname	bcity	assets

checking

acct_no	ssn	bname	balance

loan

acct_no	ssn	bname	balance

customer

ssn	cname	address

$$\pi_{\text{cname, balance}}\,(\boldsymbol{\sigma}_{\text{balance} > 1000}\,(\text{Customer} \bowtie \text{Checking}))$$

Today you will learn…

- How do databases support multiple users?
  - Today: underlying concepts
  - Future: implementation

Overview

Ensuring Data Integrity

- Issues related to preserving data integrity
  - Concurrency control
  - Crash control
- *Transactions* are a key concept at the heart of these matters
- Database is in a *consistent* state if there are no contradictions between the data within it
  - Temporary inconsistencies occur by necessity, but must not be allowed to persist
  - Example: transfer of funds between bank accounts

Transactions

Transactions are Atomic and Preserve Consistency

- A transaction is an atomic operation (unit of work) involving a series of processing steps including:
  - One or more reads and/or writes
  - Data computations can happen during a transaction, but the database is mostly concerned with reads and writes
- If the database is in a consistent state at the start of the transaction, it will be in a consistent state at the end of the transaction

ACID

- **A**tomicity: either all of the transaction completes, or none of it completes
  - If any part of the transaction fails, all effects of it must be removed from the database
- **C**onsistency: database ends the transaction in a consistent state (provided it started that way)
- **I**solation: concurrently executing transactions must be unaware of each other (as if they ran serially)
  - It should look to one as if the other has not started or has already completed
- **D**urability: a transaction's effects must persist in the database after it completes

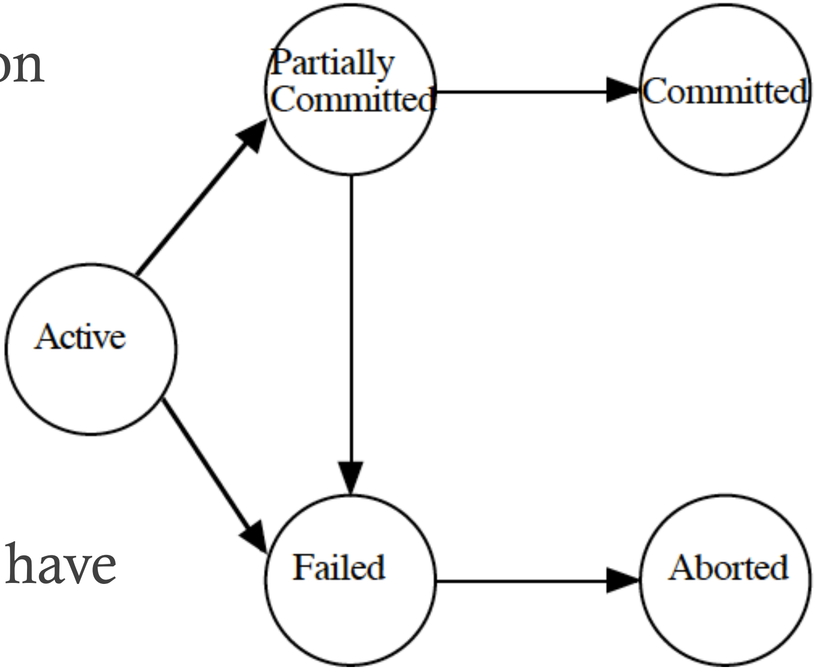
Explicit Transactions

```
BEGIN TRANSACTION


% your SQL code (SELECTs, UPDATEs, etc.)


COMMIT   % write results to the database
% or
ROLLBACK % no changes to the database
```

Implicit Transactions

- Alternatively:
  - **Autocommit:** each SQL statement in the session is treated as an individual transaction and committed upon completion
    - The default in Django
  - **Connection-based transactions:**
    - Start a transaction when the connection is opened
    - Commit the transaction when the connection is closed
    - Explicitly committing or rolling back starts a new transaction
    - Unexpected disconnection (e.g., a network error) results in a rollback

Transaction States



- Active: from the time a transaction starts until it fails or reach its last statement
- Partially committed: last statement executed, but changes to database are not yet permanent (SQL commit)
- Committed: changes to database have been made permanent
- Failed: logic error or user abort has precluded completion, and transaction's changes must be undone (SQL rollback)
- Aborted: all effects of the transaction have been removed

Schedules

- Transaction consists of a set of read and write operations
  - Other computations as well, but reads and writes are critical, since they allow one transaction to interact with another
- For two or more concurrent transactions, the relative sequence of their read and write operations constitutes a *schedule*
- Example: simultaneous $50 deposit to and $100 withdrawal from a checking account
  - In SQL, these two transactions might look like this
    - update checking_account
set balance = balance + 50
where account_no = :acct
    - update checking_account
set balance = balance – 100
where account_no = :acct
  - Each update statement actually consists of a read and a write operation

Possible Schedules (1/2)

Schedule	Deposit (T_1)	Withdrawal (T_2)	Final Balance
S_1	read(1000) write(1050)	read(1050) write(950)	950
S_2	read(1000) write(1050)	read(1000) write(900)	900
S_3	read(1000) write(1050)	read(1000) write(900)	1050

Possible Schedules (2/2)

Schedule	Deposit (T_1)	Withdrawal (T_2)	Final Balance
S_4		read(1000)	
		write(900)	
	read(900)		
	write(950)		950
S_5		read(1000)	
	read(1000)		
		write(900)	
	write(1050)		1050
S_6		read(1000)	
	read(1000)		
	write(1050)		
		write(900)	900

We Want Serial or Serializable Schedules!

- The schedules which yield the correct result are both *serial*
  - One transaction is executed in its entirety before the other starts
  - Serial schedules always lead to consistent results
    - Non-serial schedules can sometimes also yield consistent results, but determining this is not always algorithmically feasible
- To preserve data integrity, ensure that a schedule of concurrent operations is *serializable* – equivalent to some serial schedule

Result Equivalence

- Two schedules are considered *result equivalent* if operations in one schedule can be rearranged into another schedule, **without altering the resulting computation**
- Example:
    - S_1 can be converted to S_2
    - Swap order of write(A) and read(B) operations
- Note that the relative order of operations within a given transaction cannot be reordered

Schedule	T_1	T_2
S_1	read A	
		read B
	write A	
		write B
S_2	read A	
	write A	
		read B
		write B

Conflicting Operations between Transactions

- Two operations in two different transactions *conflict* if
  - They access the same data item (same column value in a single record)
    - Not same column in different records
    - Not different columns in same record
  - At least one of the operations is a write
- Changing the relative order of two conflicting operations can result in different final outcomes
- Examples:
  - Schedules 1, 2, and 3 have conflicting operations – reordering operations would lead to different outcomes
  - Schedules 4 and 5 do not have operations in conflict – no writes

Schedule	**T_1**	**T_2**
S_1	write A	
		read A
S_2	read A	
		write A
S_3	write A	
		write A
S_4	read A	
		read A
S_5		read A
	read A	

Conflict Equivalence

- Two schedules S_1 and S_2 on the same set of transactions are *conflict equivalent* if one can be transformed into the other by a series of interchanges of non-conflicting operations
- Examples
  - S_1 and S_2 are conflict equivalent
    - Access different data items
  - S_3 and S_4 are not conflict equivalent
- A schedule is *conflict serializable* if there is a serial schedule to which it is equivalent

Schedule	T_1	T_2
S_1	read A	
		read B
	write A	
		write B
S_2	read A	
	write A	
		read B
		write B
S_3	read A	
		read A
	write A	
		write B
S_4	read A	
	write A	
		read A
		write B

View Equivalence

- Two schedules S_1 and S_2 on the same set of transactions are *view equivalent* if
  - Some transaction in both schedules reads the initial value of the same data item
  - If in S_1 some transaction reads a data item that was written by another transaction, the same holds for the two transactions in S_2
  - If a transaction does the last write to some data item in S_1, it also does the last write to the same data item in S_2
- This is less strict than conflict equivalence
  - Requires that two schedules have the same outcome, but don't necessarily get there the same way (conflict equivalent)
  - Conflict equivalence implies view equivalence, but not vice versa
- A schedule is *view serializable* if it is view equivalent to some serial schedule

View Equivalence

- View equivalent schedules which are not conflict equivalent:
  - In both S_1 and S_2:
    - T_1 reads A before any other transaction has modified it
    - T_1 performs the last write on A

Schedule	T_1	T_2	T_3
S_1	read A		
		write A	
			write A
	write A		
S_2	read A		
			write A
		write A	
	write A		

Result Equivalence Doesn't Imply Conflict/View Equivalence

- Two conflict/view equivalent schedules will always produce the same final results, and so are result equivalent
  - But result equivalent schedules **aren't necessarily conflict/view equivalent**

- Example: from account deposit and withdrawal schedules
  - S_1 and S_2 produce same result, but are not conflict/view equivalent

Schedule	Deposit (T_1)	Withdrawal (T_2)	Final Balance
S_1	read(1000) write (1050)	read(1050) write(950)	950
S_4	read(900) write(950)	read(1000) write(900)	950

Equivalence Summary

- Conflict Equivalence implies View Equivalence
- View Equivalence implies Result Equivalence
- Conflict Equivalence implies Result Equivalence
- Remember that "implication" is not commutative:
  - Just because a implies b, doesn't mean b implies a.
  - For example:
    - Cat implies Animal
    - So if I know "Cookie" is a cat, I know "Cookie" is an animal
    - But just because "Fido" is an animal, "Fido" isn't necessarily a cat

Conflict/View Serializable

- A schedule is **Conflict Serializable** if it is Conflict Equivalent to a serial schedule
- A schedule is **View Serializable** if it is View Equivalent to a serial schedule

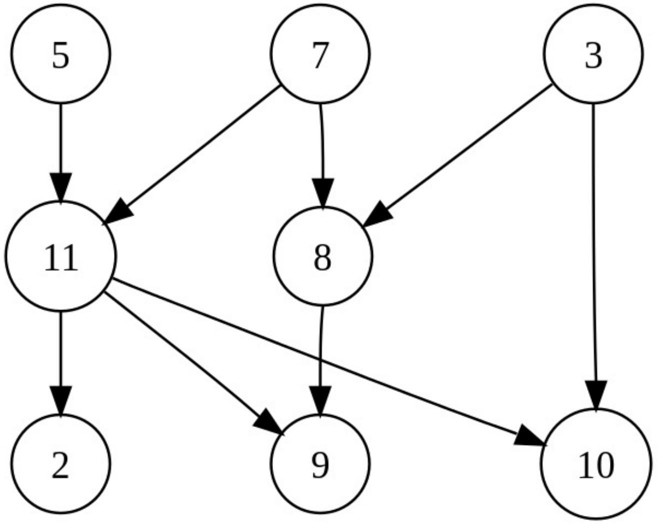
Testing for Serializability Ensures Consistency

- To ensure correctness of concurrent operations, ensure that the schedule followed is serializable
- Want to test a schedule for serializability
  - Can be very expensive to test for view serializability
  - More feasible to test for conflict serializability

Precedence Graph

- Construct a *precedence graph* of a schedule to test it for conflict serializability
  - Each transaction is a node on the precedence graph
  - There is a directed edge from Transaction_a to Transaction_b if there are conflicting operations between them – that is, at least one of the following occurs
    - T_a reads an item before T_b writes it
    - T_a writes an item before T_b reads it
    - T_a writes an item before T_b writes it
- If the resulting graph contains a cycle, the schedule is not conflict serializable
- If there are no cycles, then any topological sorting of the precedence graph will give an equivalent serial schedule

Topological Sorting

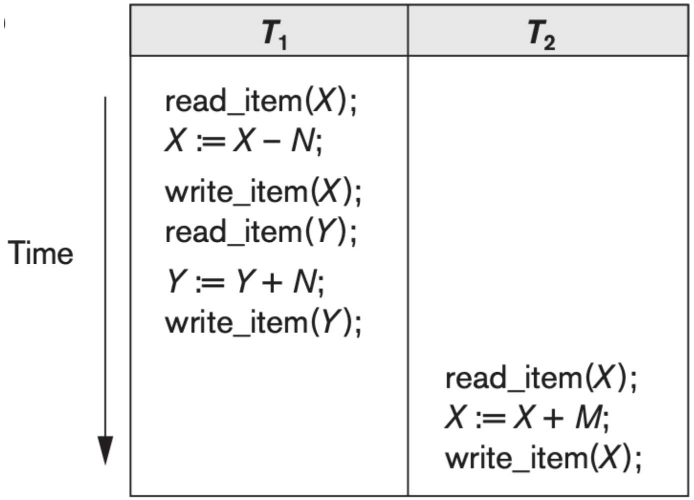


The graph shown to the left has many valid topological sorts, including:

- 5, 7, 3, 11, 8, 2, 9, 10 (visual top-to-bottom, left-to-right)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)
- 5, 7, 3, 8, 11, 10, 9, 2 (fewest edges first)
- 7, 5, 11, 3, 10, 8, 9, 2 (largest-numbered available vertex first)
- 5, 7, 11, 2, 3, 8, 9, 10 (attempting top-to-bottom, left-to-right)
- 3, 7, 8, 5, 11, 10, 2, 9 (arbitrary)

https://en.wikipedia.org/wiki/Topological_sorting

Serial

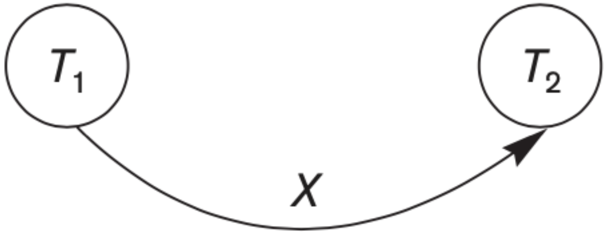


T_1	T_2

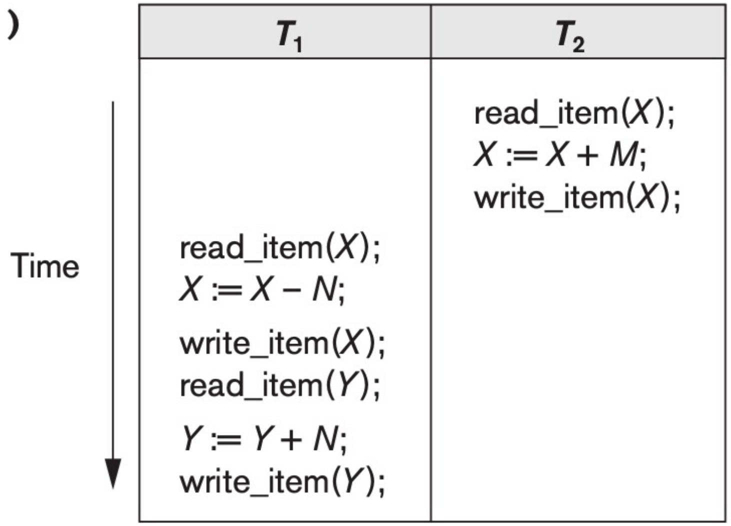
read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A



Serial

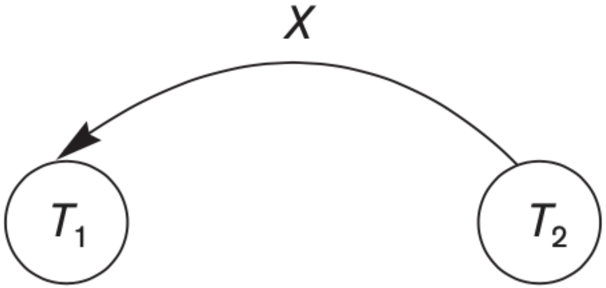


T_1	T_2

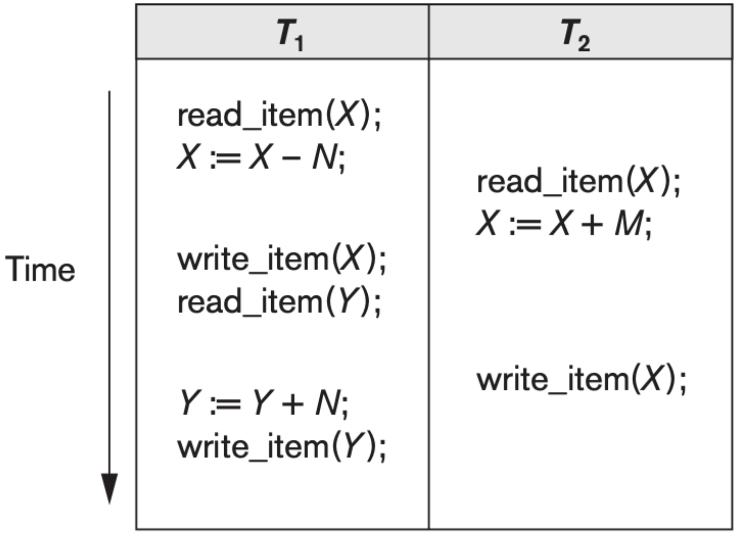
	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule B

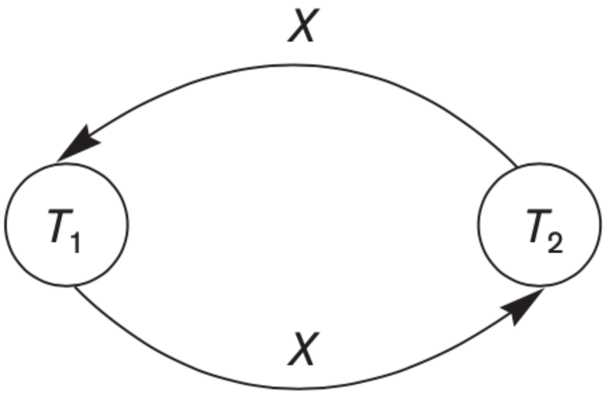


Not Conflict Serializable

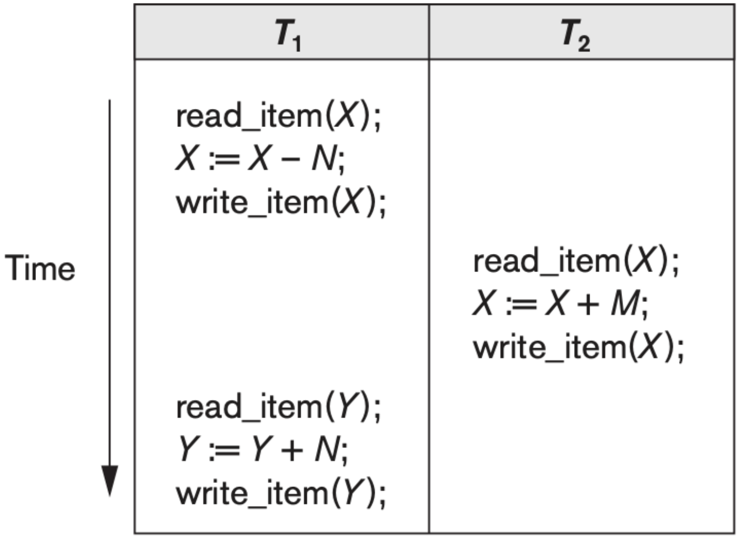


T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;
write_item(X); read_item(Y);	
	write_item(X);
$Y := Y + N$; write_item(Y);	

Schedule C



Conflict Serializable



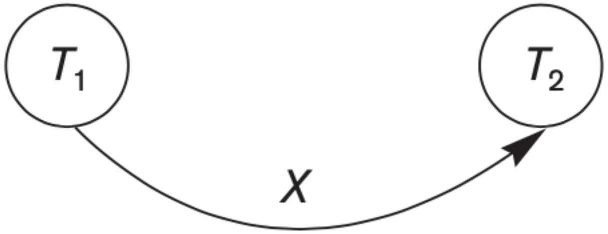
T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);
$Y := Y + N$;
write_item(Y);	

Schedule D

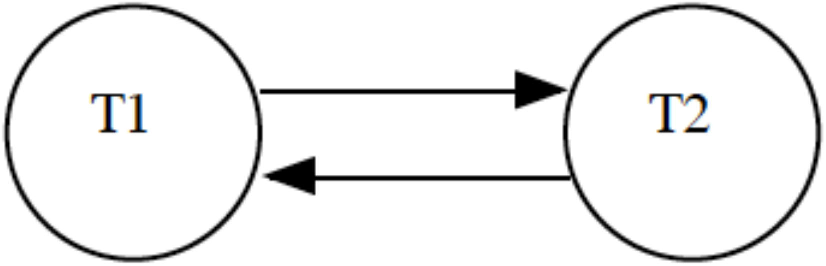


Precedence Graph Example 1

- Consider this schedule:

Deposit (T_1)	**Withdrawal (T_2)**	**Final Balance**
read savings(1000)		
	read savings(1000)	
write savings(1050)		
	write savings(900)	savings(900)

  - T_1 must read before T_2 writes
  - T_2 must read before T_1 writes
- Yields a cyclical precedence graph
  - Schedule is not serializable



Precedence Graph Example 2

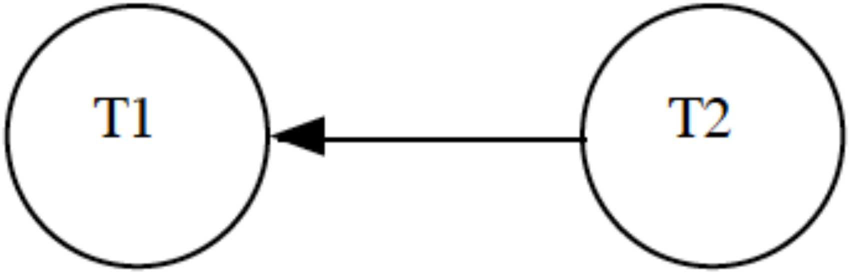
- Consider a transfer of \$50 from a savings account (with a \$2000 starting balance) to a checking account that occurs at the same time as a \$100 checking account withdrawal via the following schedule

Transfer (T_1)	Withdrawal (T_2)	Final Balances
read savings (2000)		
	read checking (1000)	
write savings (1950)		1950 (savings)
	write checking (900)	
read checking (900)		
write checking (950)		950 (checking)

- Note the following conflicting operations in this schedule:
  - T_2 must read checking before T_1 writes to checking
  - T_1 must read checking after T_2 writes to checking

Precedence Graph Example 2 (Continued)

- Yields this precedence graph
  - Acyclic – indicates a serializable schedule
  - T_2 can be done before T_1
  - Leads to the following conflict equivalent serial schedule



Transfer (T_1)	Withdrawal (T_2)	Final Balances
	read checking (1000)	
	write checking (900)	
read savings (2000)		
write savings (1950)		1950 (savings)
read checking (900)		
write checking (950)		950 (checking)

Transaction Recoverability

- Schedules must not only serializable, but *recoverable*
  - Unrecoverable schedules can lead to inconsistencies
  - A transaction T_2 must not commit until any transaction T_1 which produces data used by T_2 commits
    - If T_1 fails, then T_2 must also fail
- Avoid *cascading rollback* – possibility of chain of failed transactions
  - T_2 reads data from T_1, T_3 reads data from T_2 T_4 reads data from T_3
  - If T_1 fails – T_2, T_3, and T_4 must also fail
- Producing only *cascadeless schedules* is desirable
  - No transaction T_2 is allowed to read a value written by another transaction T_1 until T_1 has fully committed
    - T_2 must wait until T_1 commits or fails (in which the previous value of the uncommitted item is used)

Summary

- Multiple transactions can conflict with each other
  - Conflicts be efficiently detected using precedence graphs
  - Non-conflicting transactions are "Conflict Serializable"
- When there is a conflict, one of the transactions must be rolled back
  - Crash recovery must be aware of these ordinary transaction failures
- Next: different techniques can be used to implement crash recovery