CSAI 302

Advanced Database Systems

Lec 07

Transaction Processing and Schedules

Introduction

- Transactions (informally):
  - mechanism for managing logical units of data processing: **independent on others**, **all or nothing**
  - Examples:
    - a single data retrieval query
    - a sequence of data manipulation queries that should be executed together
- Transaction management (or processing) systems:
  - systems with **large databases** and many **concurrent users** require **high availability** and **fast response** time
  - Examples:
    - banking, airline booking, online retail, stocks

Fund Transfer Example

- Transfer $50 from account A to account B:
  1. read(A)
  2. A := A – 50
  3. write(A)
  4. read(B)
  5. B := B + 50
  6. write(B)

Failures, such as:
- **hardware failures**
- **system crashes**

Concurrent execution of multiple transactions

Fund Transfer Example

- Transfer $50 from account A to account B:
  - 1. read(A)
  - 2. A := A – 50
  - 3. write(A)
  - 4. read(B)
  - 5. B := B + 50
  - 6. write(B)

- If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
- The system should ensure that updates of a partially executed transaction are not reflected in the database

Atomicity requirement

Fund Transfer Example

- Transfer $50 from account A to account B:
  - 1. read(A)
  - 2. A := A – 50
  - 3. write(A)
  - 4. read(B)
  - 5. B := B + 50
  - 6. write(B)

- Once the user has been **notified** that the transaction **has completed**
  - The transfer of the $50 has taken place
- the updates to the database by the transaction must persist even if there are software or hardware failures.

Durability requirement

Fund Transfer Example

- Transfer $50 from account A to account B:
  - 1. read(A)
  - 2. A := A – 50
  - 3. write(A)
  - 4. read(B)
  - 5. B := B + 50
  - 6. write(B)

- Sum of A and B **is unchanged** by the execution of the transaction
  - A transaction must see a **consistent database**.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent

Consistency requirement

Fund Transfer Example

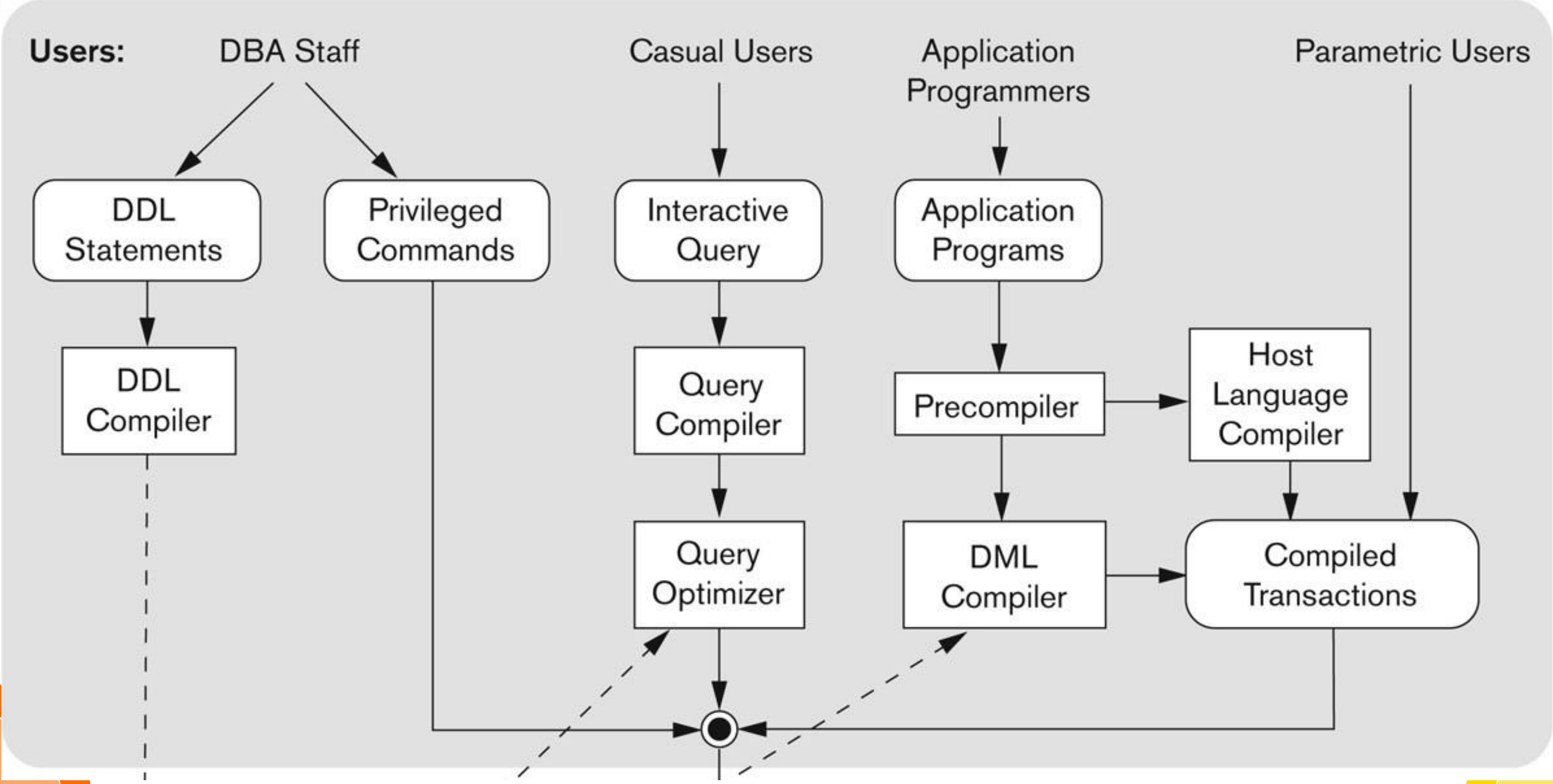
- Transfer \$50 from account A to account B:
  - 1. read(A)
  - 2. A := A – 50
  - 3. write(A)
  - 4. read(B)
  - 5. B := B + 50
  - 6. write(B)
- if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database
- T2
  - **read(A), read(B), print(A+B)**
- it will see an inconsistent database
- (the sum $A + B$ will be less than it should be).

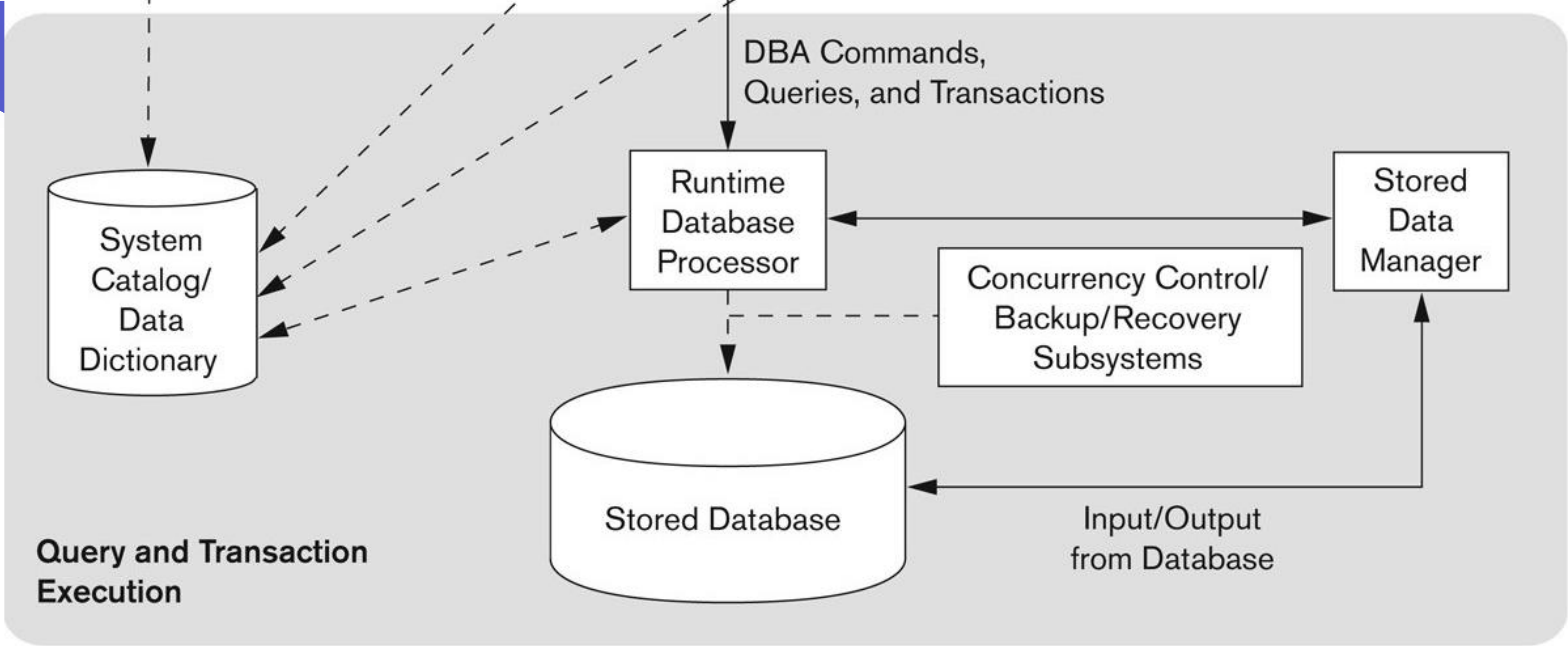
Isolation requirement

Single-user vs multi-user

- Single-user DBMS:
  - at most one user at a time (usually, personal computer)
- Multi-user DBMS:
  - many users (processes with own computation) with concurrent access to the same data
    - (usually, servers with many CPUs, but may be handled by one CPU with interleaving concurrency)
- Both need transaction management, with:
  - concurrency control
    - (transactions independent of each other)
  - fail recovery
    - (each transaction executed all or nothing)

The place of transaction management




DBA Commands,
Queries, and Transactions
System
Catalog/
Data
Dictionary
Runtime
Database
Processor
Stored
Data
Manager
Concurrency Control/
Backup/Recovery
Subsystems
Stored Database
Input/Output
from Database
Query and Transaction
Execution

Transaction idea

- Transaction is a program that forms a logical unit of database processing
  - has its own memory and computation ability
  - includes one or more access operations to (shared) database (e.g., retrieval, insertion, deletion)
- boundaries can be specified by **begin** and **end** statements
  - if something goes wrong on the way, effects ***roll back***
- an application program may have several transactions
  - may be **read-only** or **read-write**

Main transaction operations

- **read(X)**
  - reads an item named X from the global database into a local program variable named X
    - includes finding the address of the block on the disk (or in cache) with X , and copying to a main memory buffer
- **write(X)**
  - writes the value of local program variable named X into the global database item named X
    - includes finding the address of the block on the disk (or in cache) with X , read it to the local memory buffer, modify it, and write it back (to the disk or cache)
- **Program local operations**
  - (for example, update X := X + 50)

Example transactions

- Let X and Y be the numbers of reserved seats in two flights (stored in a database)
- Transaction T1 transfers N reservations from X to Y
- Transaction T2 reserves M seats in X
- Important:
  - updates are local, the database is not updated until the new value is written
  - some commands are often omitted if they are not relevant (both local and transaction management)

T_1
read(X);
$X := X - N$;
write(X);
read(Y);
$Y := Y + N$;
write(Y);

T_2
read(X);
$X := X + M$;
write(X);

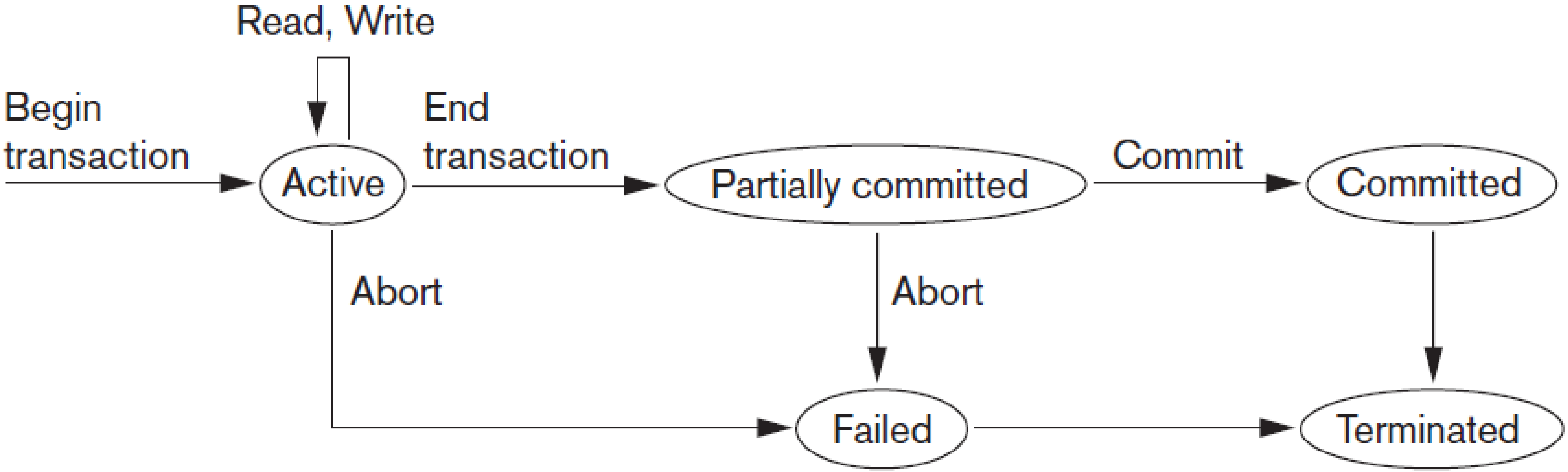
Types of failures

Computer failure (system crash)	• hardware, software, network errors
Transaction failure	• division by zero, integrity constraint violation, user interrupt, etc.
Local transaction errors	• no data found, programmed exception, etc.
Concurrency control enforcement	• serializability violation, deadlock resolving, etc.
Disk failure	• errors with disk reads or writes
Physical problems	• power cut, fire, catastrophe, etc.

Transaction State

- Active
  - the initial state; the transaction stays in this state while it is executing
- Partially committed
  - after the final statement has been executed.
- Failed
  - after the discovery that normal execution can no longer proceed.
- Aborted
  - after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
  - Two options after it has been aborted:
    - Restart the transaction (*only if no internal logical error*)
    - Kill the transaction
- Committed
  - after successful completion.

Transaction State (Cont.)



Desirable properties
ACID principles

ACID properties

Atomicity	Consistency	Isolation	Durability
Transaction performed in its entirety or not at all	The database should always remain consistent	Transaction should not interfere with other transactions	Changes of committed transactions must persist
ensured by transaction **recovery subsystem**	ensured by transaction **recovery subsystem**	ensured by the **concurrency control subsystem**	ensured by transaction **recovery subsystem**

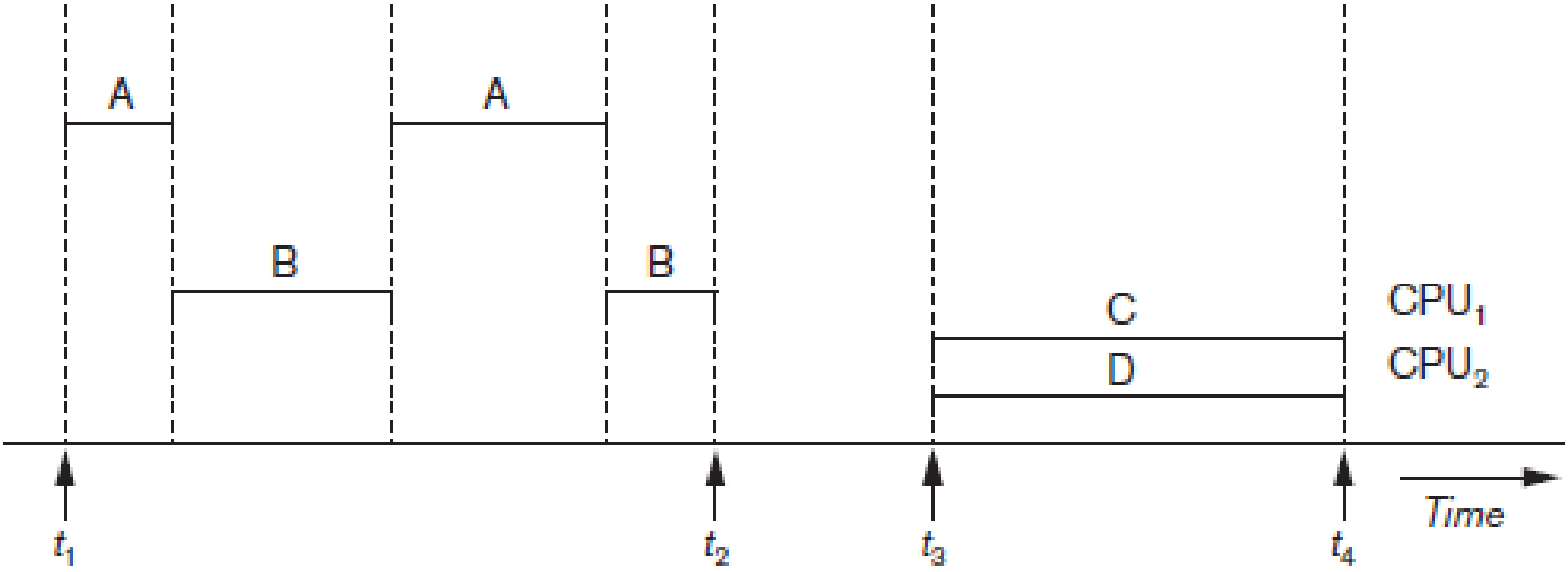
Example

- Consider two transactions (Xacts):
  - **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 credits both accounts with 6% interest.
- Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2?
  - T1 ; T2 (A=1166,B=954)
  - T2 ; T1 (A=1160,B=960)
  - In either case, A+B = $2000 *1.06 = $2120
  - There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.

Example (Contd.)

- Consider a possible interleaved schedule:
- **T1: A=A+100, B=B-100**
- **T2: A=1.06*A, B=1.06*B**
  - This is OK (same as T1;T2). But what about:
- **T1: A=A+100, B=B-100**
- **T2: A=1.06*A, B=1.06*B**
  - *Result: A=1166, B=960; A+B = 2126, bank loses $6 !*
- 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)**

Interleaved vs. Parallel processing



Transactions and schedules

Transactions

- Transaction is a sequence of operations, an atomic unit of work
- Structure of a **committed** (successful) transaction:
  1. **begin** ← marks the beginning of transaction execution
  2. one or several **read(X)**, **write(X)**, local operations (e.g., var updates), transaction control operations (e.g., locks), etc.
  3. **End** ← mark the end of transaction execution
  4. one or several operations checking consistency, serializability, etc.
  5. **commit** ← successful completion, changes cannot be undone
- Structure of an **aborted** (unsuccessful) transaction: same beginning, but ends at any point with
  - N. **abort** ← unsuccessful completion, all changes must be undone
- A **partial** transaction:
  - a beginning of one above ('waiting' for next operation)

Schedules

- Schedule (or history, execution plan) S of transactions $T_1, \ldots, T_n$: a (total) ordering of the operations of $T_1, \ldots, T_n$
  - operations of different transactions can interleave
  - operations of each Ti are in the same order as in Ti

Schedules

- Example:
  - Sa: $r1(X), r2(X), w1(X), r1(Y), w2(X), w1(Y)$

Time

T_1	T_2
read(X); $X := X - N$;	
	read(X); $X := X + M$;
write(X); read(Y);	
	write(X);
$Y := Y + N$; write(Y);	

- Example (complete):
  - Sb : $r1(X), w1(X), r2(X), w2(X), r1(Y), a1, c2$

Complete Schedule

- all $T_1, \ldots, T_n$ are committed or aborted
  - A transaction that **successfully** completes its execution will have a **commit** instructions as the last statement
    - By default, transaction assumed to execute commit instruction as its last step
  - A transaction that **fails** to successfully complete its execution will have an **abort** instruction as the last statement

Example

- Let
  - T1 transfer $50 from A to B,
- and
  - T2 transfer 10% of the balance from A to B.

Schedule 1

- A serial schedule in which T1 is followed by T2

T_1	T_2
read (A)	
A := A – 50	
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	
	read (A)
	$temp$:= A * 0.1
	A := A - $temp$
	write (A)
	read (B)
	B := B + $temp$
	write (B)
	commit

Schedule 2

- A serial schedule where T2 is followed by T1

T_1	T_2
	read (A)
	$temp$:= A * 0.1
	A := A - $temp$
	write (A)
	read (B)
	B := B + $temp$
	write (B)
	commit
read (A)	
A := A – 50	
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	

Schedule 3

- This schedule is not a serial schedule, but it is equivalent to Schedule 1
  - In Schedules 1, 2 and 3, the sum A + B is preserved.

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

Schedule 4

- The following concurrent schedule does not preserve the value of (A + B).

T_1	T_2
read (A)	
A := A – 50	
	read (A)
	$temp$:= A * 0.1
	A := A - $temp$
	write (A)
	read (B)
write (A)	
read (B)	
B := B + 50	
write (B)	
commit	
	B := B + $temp$
	write (B)
	commit

Serializability

Serializability

- Each transaction should preserve database **consistency**.
  - ★ ***Serial execution*** of a set of transactions preserves database consistency.
- A (possibly ***concurrent***) schedule is ***serializable*** if it is *equivalent* to a serial schedule.

Serializable schedules

Time

T_1	T_2
read(X); $X := X - N$;	
	read(X); $X := X + M$;
write(X); read(Y);	
	write(X);
$Y := Y + N$; write(Y);	

Schedule C

Time

T_1	T_2

read(X);
$X := X - N$;
write(X);	

	read(X);
	$X := X + M$;
	write(X);

read(Y);
$Y := Y + N$;
write(Y);	

Schedule D

- **C** not serializable

D serializable

Conflicting Instructions

- Instructions ***Ii*** and ***Ij*** of transactions ***Ti*** and ***Tj*** respectively,
- **conflict**
  - ***if and only if*** there exists some item **Q** accessed by both ***Ii*** and ***Ij***,
  - and ***at least*** one of these instructions **wrote** **Q**.

Conflicting Instructions

- **li = read(Q), lj = read(Q).** don't conflict.
- **li = read(Q), lj = write(Q).** They conflict
  - ★ r1(X), r2(X), w1(X), r1(Y), w2(X), w1(Y)
- **li = write(Q), lj = read(Q).** They conflict
  - ★ r1(X), w1(X), r2(X), w2(X), r1(Y), a1
- **li = write(Q), lj = write(Q).** They conflict
  - ★ r1(X), w1(X), r2(X), w2(X), r1(Y), a1

Conflicting Instructions

- A conflict between I_i and I_j forces a (logical) temporal order between them.
- If I_i and I_j are consecutive in a schedule and they do not conflict,
  - their results would remain the same even if they had been interchanged in the schedule.

Serializability Types

Different forms of schedule equivalence give the notions of:

- Conflict serializability
- View serializability

Conflict Serializability

- If a schedule **S** can be transformed into a schedule **S'** by a series of swaps of non-conflicting instructions, we say that **S** and **S'** are *conflict equivalent*.
- We say that a schedule **S** is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict equivalent

- Two schedules are **conflict equivalent** if
  - they are schedules of the same transactions
  - if the **relative order** of every conflict (read-write, write-write) is the **same** in both schedules

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore **Schedule 3** is conflict serializable.

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T3, T4 >, or the serial schedule < T4, T3 >.

Conflict Serializability (Cont.)

Time

T_1	T_2
read(X); $X := X - N$;	
	read(X); $X := X + M$;
write(X); read(Y);	
	write(X);
$Y := Y + N$; write(Y);	

Schedule C

Time

T_1	T_2

read(X);
$X := X - N$;
write(X);	

	read(X);
	$X := X + M$;
	write(X);

read(Y);
$Y := Y + N$;
write(Y);	

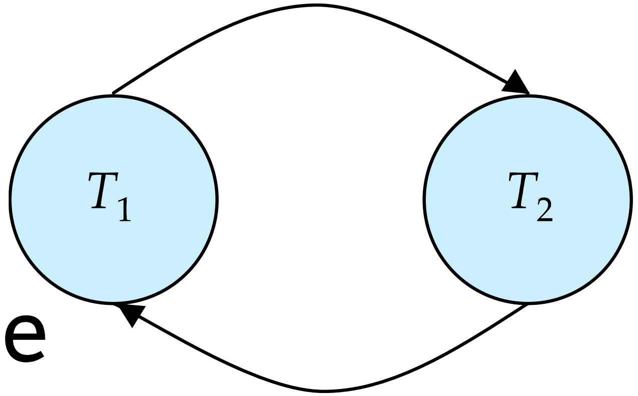
Schedule D

Schedule C is **not (conflict-)serializable**

Schedule D is **serializable** (equivalent to $T_1; T_2$)

Testing for Serializability

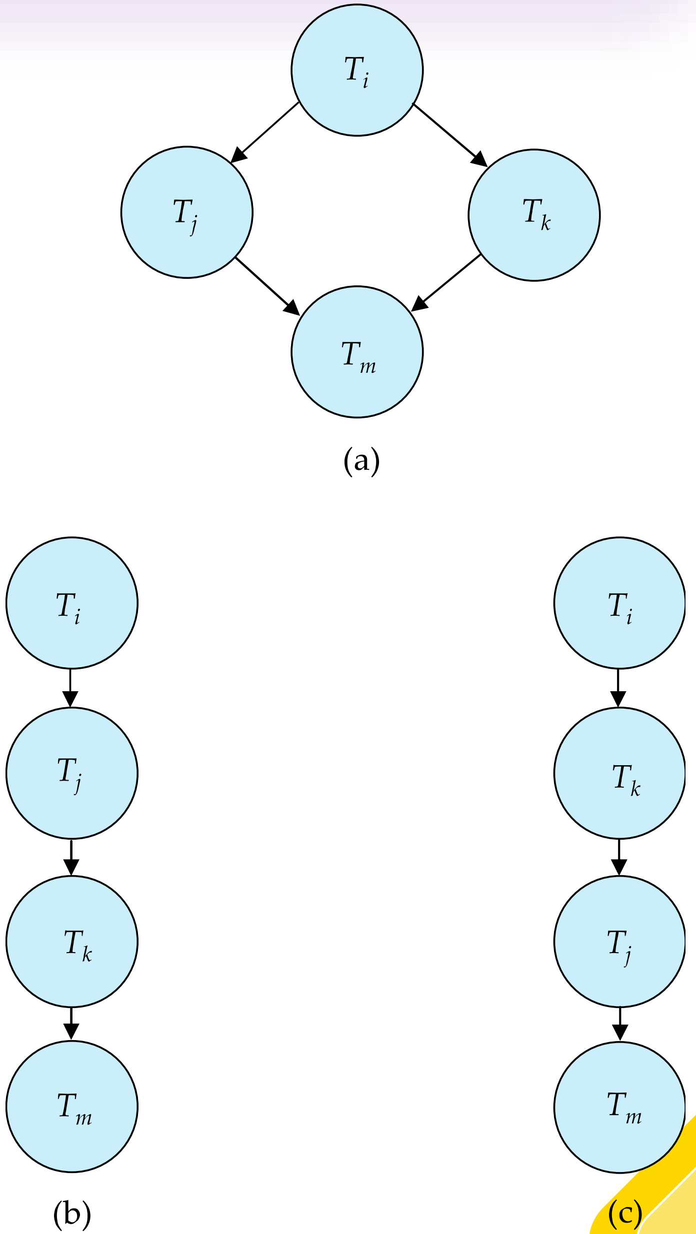
- Consider some schedule of a set of transactions ***T1***, ***T2***, ..., ***Tn***
- **Precedence graph**
  - a direct graph where the vertices are the transactions.



- We draw an arc from ***Ti*** to ***Tj*** if the two transactions conflict, and ***Ti*** accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.

Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is ***acyclic***.
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be
    $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?



conflict serializable

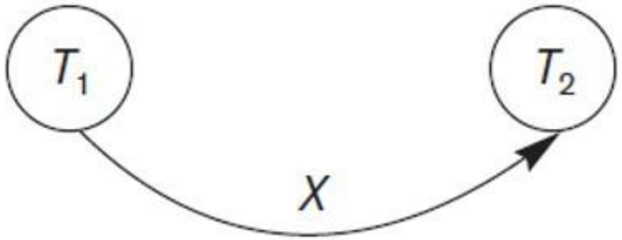
Time ↓	T_1	T_2
	read(X);	
	$X := X - N$;	
	write(X);	
	read(Y);	
	$Y := Y + N$;	
	write(Y);	
		read(X);
		$X := X + M$;
		write(X);

Schedule A

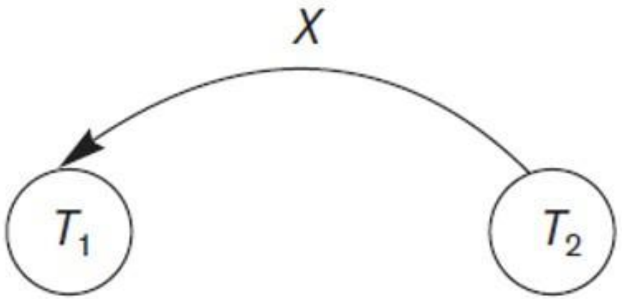
Time ↓	T_1	T_2
		read(X);
		$X := X + M$;
		write(X);
	read(X);	
	$X := X - N$;	
	write(X);	
	read(Y);	
	$Y := Y + N$;	
	write(Y);	

Schedule B

(a)



(b)



Example 1

Time

T_1	T_2
read(X); $X := X - N$;	
	read(X); $X := X + M$;
write(X); read(Y);	
	write(X);
$Y := Y + N$; write(Y);	

Schedule C

Time

T_1	T_2

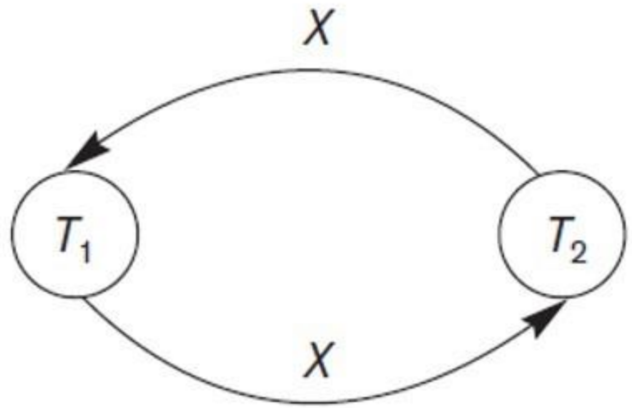
read(X);
$X := X - N$;
write(X);	

	read(X);
	$X := X + M$;
	write(X);

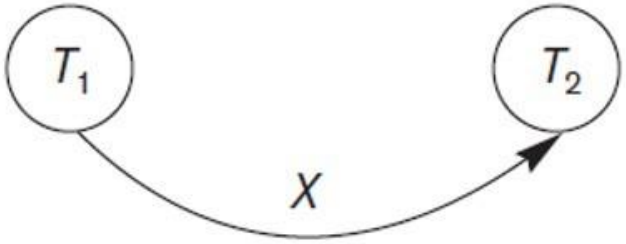
read(Y);
$Y := Y + N$;
write(Y);	

Schedule D

(c)



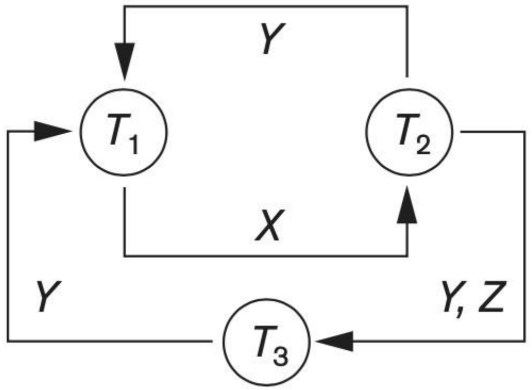
(d)



Example 2

Time

Transaction T_1	Transaction T_2	Transaction T_3
	read(Z);	
	read(Y);	
	write(Y);	
		read(Y);
		read(Z);
read(X);		
write(X);		
		write(Y);
		write(Z);
	read(X);	
read(Y);		
write(Y);		
	write(X);	



Equivalent serial schedules

None

Reason

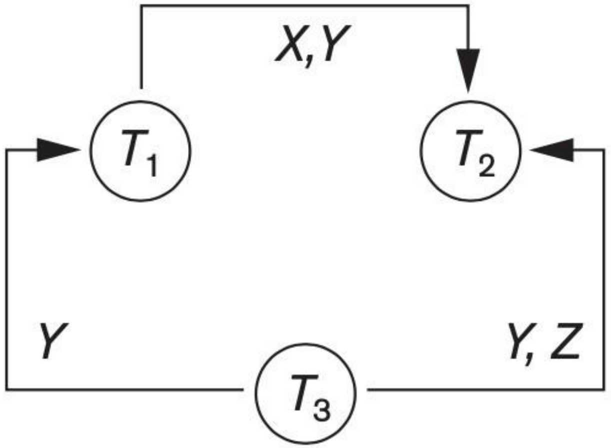
Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

Cycle $X(T_1 \rightarrow T_2), YZ\ (T_2 \rightarrow T_3), Y(T_3 \rightarrow T_1)$

Example 3

Time	Transaction T_1	Transaction T_2	Transaction T_3
↓			read(Y); read(Z);
	read(X); write(X);		
			write(Y); write(Z);
		read(Z);	
	read(Y); write(Y);		

		read(Y);
		write(Y);
		read(X);
		write(X);	



Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

View Serializability

- Let **S** and **S'** be two schedules with the same set of transactions.
- **S** and **S'** are **view equivalent** if the following three conditions are met, for each data item **Q**.

1. If in **S**, transaction **Ti** **reads** the initial value of Q,
   then in **S'** transaction **Ti** must **read** the initial value of Q.
2. If in **S** transaction **Ti** executes **read**(Q), and that value was produced by transaction **Tj** ,
   then in **S'** transaction **Ti** must **read** the value of Q that was produced by the same **write**(Q) of **Tj** .
3. The transaction that performs the final **write**(Q) operation in **S** must also perform the final **write**(Q) operation in **S'**.

View-serializability

- serializability based on view equivalence
- Each conflict-serializable is view-serializable
- Each view-serializable is semantic-serializable (so result-serializable)
- Difficult to check view-equivalence (NP-complete)

View Serializability (Cont.)

- A schedule S is **view serializable** if it is **view-equivalent** to a serial schedule.
- Below is a schedule which is view-serializable but not conflict serializable.

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

Test for View Serializability

- The precedence graph test for **conflict serializability** cannot be used directly to test for **view serializability**.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view-serializable falls in the class of NP-complete problems.
  - Thus, existence of an efficient algorithm is extremely unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Serializability in concurrency control

- Every serial schedule is serializable, but not other way round
- Serializable schedules give benefit of concurrent execution without giving up any correctness
- Difficult to test for serializability in practice (even conflict-serializability)
  - system load, time of transaction submission, and process priority affect ordering of operations
  - often, we need to ensure serializability before the transactions complete
- DBMSs enforce concurrency control protocols that ensure serializability

Recoverability

- For **durability**, we need to be able to **recover** schedules from transaction and system failures
- Schedules with respect to recoverability, are classified into:
  - impossible to recover
  - possible to recover
  - recoverable
  - easy to recover
  - cascadeless, strict

Recoverable Schedules

- Need to address the effect of transaction failures on concurrently running transactions.
- <u>**Recoverable schedule**</u>
  - if a transaction **Tj** reads a data item previously written by a transaction **Ti** ,
  - then the **commit** operation of **Ti** appears before the **commit** operation of **Tj**.

Recoverable Schedules

- The following schedule is ***not recoverable***
- If T8 should **abort**, T9 would have **read** (and possibly shown to the user) an **inconsistent** database state.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

Examples

- r1(X), r2(X), w1(X), r1(Y), w2(X), c2, w1(Y), c1
  - recoverable
- r1(X), w1(X), r2(X), r1(Y), w2(X), w1(Y), c2, a1
  - non-recoverable
- r1(X), w1(X), r2(X), r1(Y), w2(X), w1(Y), a1, c2
  - still non-recoverable
- r1(X), w1(X), r2(X), r1(Y), w2(X), w1(Y), a1, a2
  - recoverable

Cascading Rollbacks

- A single transaction failure leads to a series of transaction rollbacks.
  - Can lead to the undoing of a significant amount of work
- Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)
  - If T10 fails, T11 and T12 must also be rolled back.

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Cascadeless schedules

- Recoverable schedules may need cascading aborts (cascading rollbacks) to recover
  - an (non-committed) transaction has to abort since it has read from another failed transaction
  - Example
  - **r1(X), w1(X), r2(X), r1(Y), w2(X), w1(Y), a1, a2**
- This may affect many transactions, and rollbacks may be expensive
- ***Cascadeless schedule idea:* no cascade rollbacks**

Cascadeless schedules

- Every transaction in the schedule reads only from **committed** transactions
  - (i.e., for each r(X), X can be previously written by the same transaction, written by another **committed** transaction or have not been written before)
- Example:
  - **r1(X), w1(X), r1(Y), r2(Z), w1(Y), c1, w2(Z), r2(X), w2(X), c2**

Strict schedules

- Cascadeless schedules may still have fairy complicated recovery protocols, because we cannot just restore the values of all writes of an aborted transaction
  - Example: **w1(X), w2(X), a1**
  - This may all affect many transactions (no cascades though)
- Strict schedule: no transaction can read or write an item X until the previous write of X is committed or aborted
- Examples:
  - **w1(X), c1, w2(X)**
  - **w1(X), a1, w2(X)**

Recovery hierarchy

- Every **strict** schedule is **cascadeless**
- Every **cascadeless** is **recoverable**

THANK YOU