南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

Transaction Processing

南方科技大学
唐　博
tangb3@sustech.edu.cn

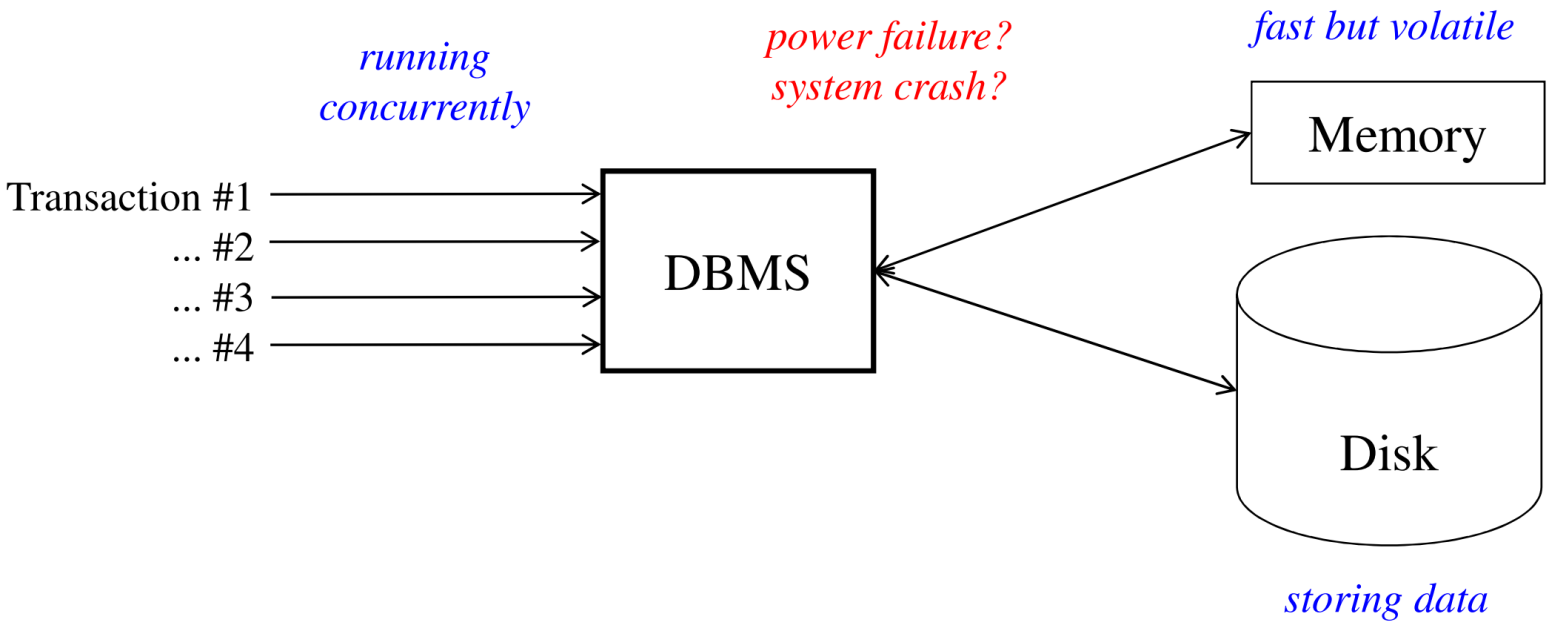
Outline

- **Basic concepts** on transaction processing
- What is a serializable schedule?
- How to test for a serializable schedule?
- What is a recoverable schedule?

Outline

- ✓ **Basic concepts** on transaction processing
- ❖ What is a serializable schedule?
- ❖ How to test for a serializable schedule?
- ❖ What is a recoverable schedule?

Transaction Processing



- What is a transaction?
- Which properties should a DBMS guarantee?

  ACID properties

  (Atomicity, Consistency, Isolation, Durability)

What is a Transaction?

- **Transaction:** a *unit* of program that *accesses* and *updates* data items

Transaction to transfer $v
from account A to account B:

1. **read**(A)
2. $A := A - v$
3. **write**(A)
4. **read**(B)
5. $B := B + v$
6. **write**(B)

- **Operations**
  - read (X), write (X) (our focus)
  - arithmetic operations (we do not care)
  - commit or abort at end of transaction (to be discussed later)

Transaction in SQL

- SQL extensions to support transactions
- E.g., Transact-SQL (Microsoft, Sybase), PL/SQL (Oracle)
  - Can declare variables
  - Support flow control statements (e.g., if, else, while)
  - Support multiple SQL statements (e.g., SELECT, INSERT, DELETE, UPDATE)
  - COMMIT and ROLLBACK statements
  - https://en.wikipedia.org/wiki/Transact-SQL
- For simplicity, we write transactions like on the previous page

ACID properties: Consistency

- **Consistency:** Each single transaction preserves the consistency of the database
  - The consistency requirement depends on the application
  - Example: a fund transfer transaction should not create / destroy money
  - [Manual] Application programmers should avoid bugs in transactions
  - [Automatic] Check consistency at runtime by using integrity constraints
    - But expensive!

A *buggy* transaction to transfer $\$x$ from account A to account B:

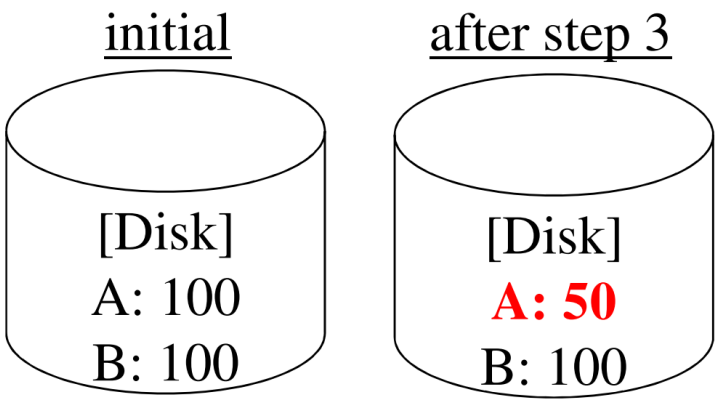
1. **read**(A)
2. $A := A - x$
3. **write**(A)
4. **read**(B)
5. $B := B + 2x$
6. **write**(B)

ACID properties: Atomicity

- **Atomicity:** Either all operations or none operations of the transaction are reflected in the database
  - Suppose that the system updates data items in the disk immediately
  - What happens if the system **fails** at this point?
    - E.g., power failure

Transaction to 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)



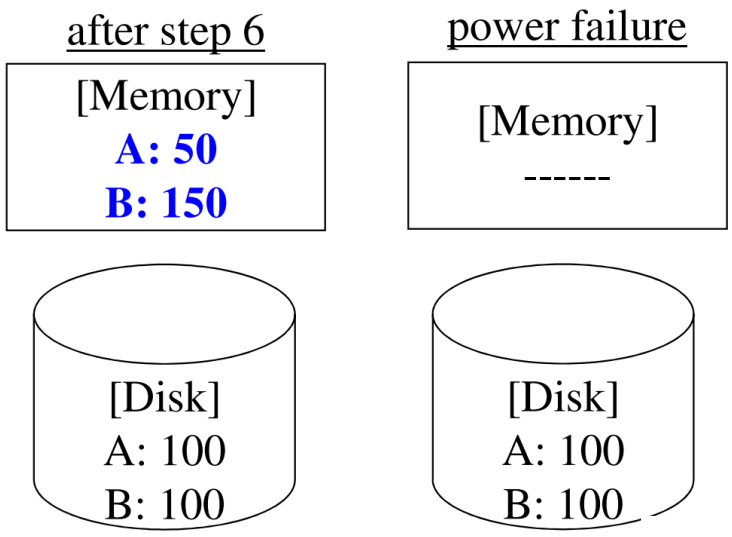
ACID properties: Durability

- **Durability:** After a transaction completes successfully, the changes made to the database must persist, even if there are system failures
  - Suppose that the system caches data items in the main memory
  - After the transaction has completed,

    what happens if the system **fails** at this point?

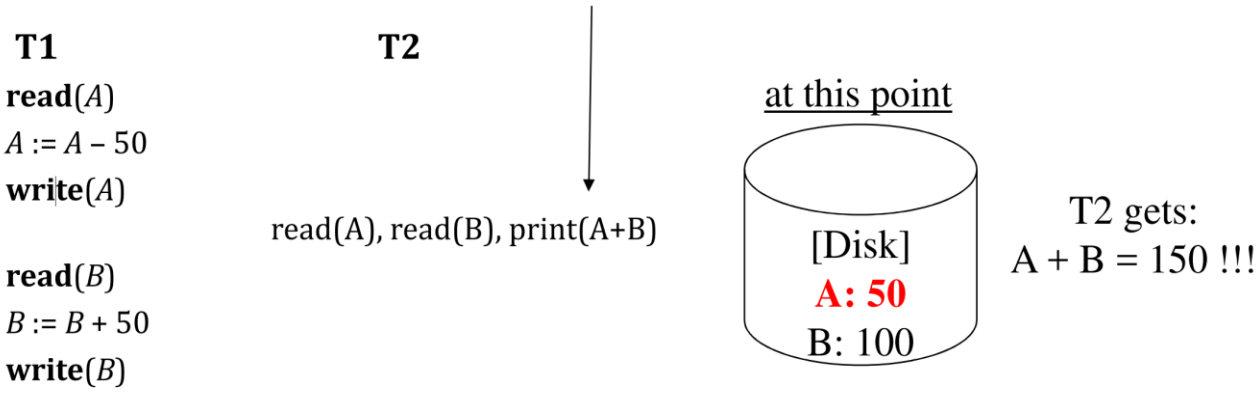
Transaction to transfer $50 from account A to account B:

1. **read**(A)
2. $A := A - 50$
3. **write**(A)
4. **read**(B)
5. $B := B + 50$
6. **write**(B)

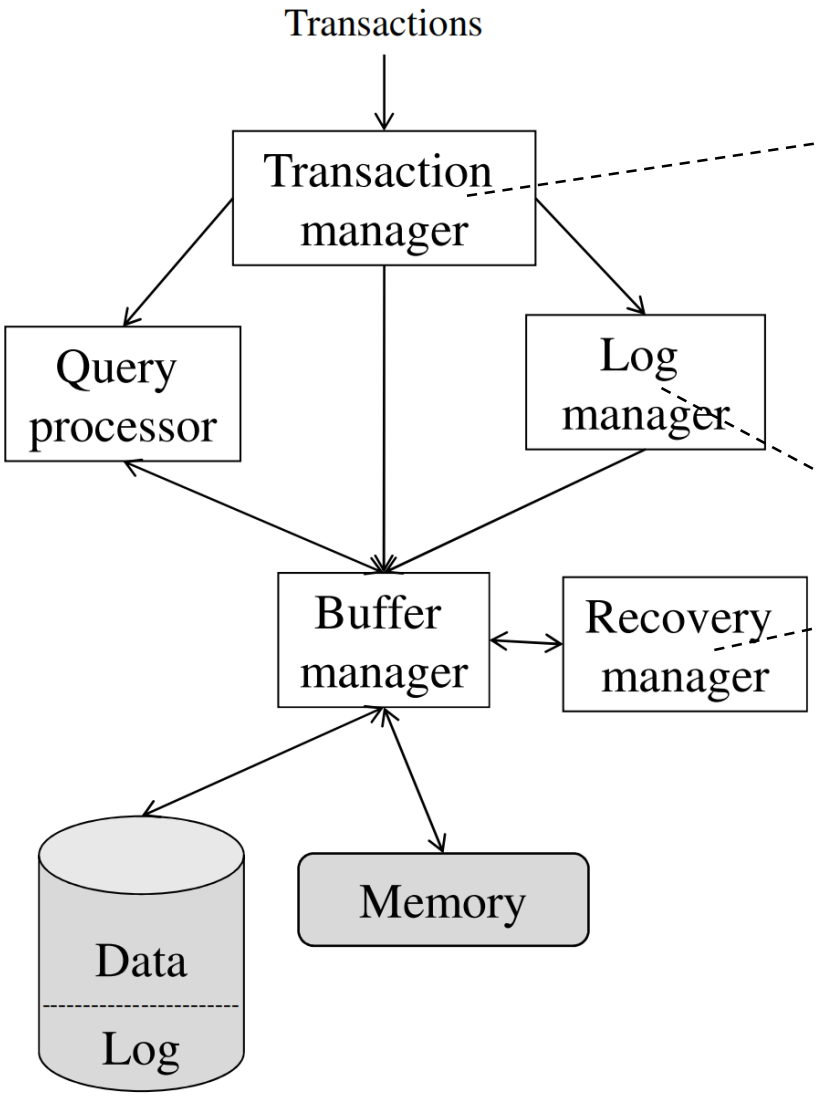


ACID properties: Isolation

- Advantages for DBMS to run multiple transactions concurrently
  - **Better utilization of CPU and disk**
    - a transaction is using the CPU while another is using the disk
  - **Reduced average response time** for transactions:
    - short transactions need not wait behind long ones
- **Isolation:** Each transaction cannot see any intermediate transaction result from concurrently executing transactions
  - Example: Transaction T2 can access the partially updated database (caused by T1) → T2 may see an inconsistent database



DBMS components for transaction processing



- Coordinates transactions that operate in parallel and access shared data ('*I*')
  - concurrency control schemes
- Ensure that failures do not corrupt persistent data ('*A*' , '*D*')
  - logging and recovery

Transaction Operations

- Data access operations: *read*, *write*
  - Advanced operations (e.g., *increment* and *decrement*) can be executed as *atomic* actions

- **Transaction operations:** ***Start***, ***Commit***, ***Abort***
  - **Start:** begins a new transaction
  - **Commit:** terminates a transaction successfully, and all of its effects should be made permanent
  - **Abort:** terminates a transaction abnormally and all of its effects should be eliminated
    - We roll back an aborted transaction and undo its effects

Commit/Abort Example

Transfer

```
Start;
input(from-ac, to-ac, amount);
temp = read(Account[from-ac]);
if (temp < amount) {
   output("insufficient fund");
   abort;
} else {
   write(Account[from-ac], temp – amount);
   temp = read(Account[to-ac]);
   write(Account[to-ac], temp + amount);
   commit;
   output("transfer completed");
}
```

When a transaction fails to complete its execution

When a transaction successfully completes its execution

// Note: we have omitted other error checking in this example, e.g., input check.

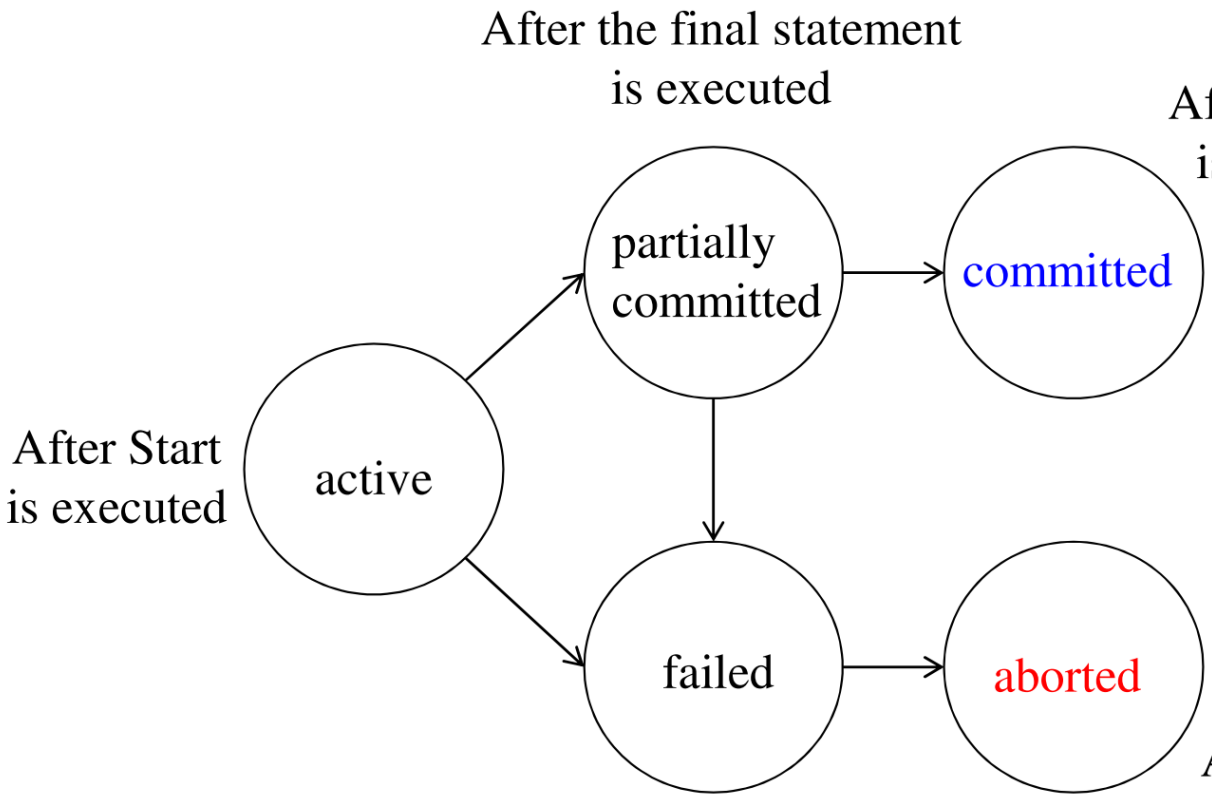
More on Abort and Commit

- Three reasons to abort a transaction:
  - Its own logic (e.g., insufficient fund in "from-ac")
  - Caused by DBMS due to system failures
  - Caused by DBMS due to concurrency control (e.g., the transaction reads an uncommitted value)

- To ensure **atomicity**, we must **undo** any changes caused by the aborted transaction
  - Also, the effect of an aborted transaction must not be seen by other transactions

- Executing a transaction's **commit** guarantees that
  - The transaction's effects will not be undone in future and
  - Such effects will survive the system's failures in future

Transaction States



A transaction is said to be *terminated* if it either has committed or is aborted

Aborting a Transaction

- Two ways to deal with an aborted transaction:
  - *Restart:* if the abort is due to hardware failure or concurrency control, the system could restart the transaction after the system recovers
  - *Kill:* if the abort is due to logical error, the transaction is discarded

Outline

- ✓ **Basic concepts** on transaction processing
- ✓ What is a serializable schedule?
- ❖ How to test for a serializable schedule?
- ❖ What is a recoverable schedule?

Schedules

- **Schedule** – it specifies the execution order of instructions of concurrent transactions
  - must contain all instructions of those transactions
  - must preserve the order of instructions within each individual transaction

Transactions

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

a schedule

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

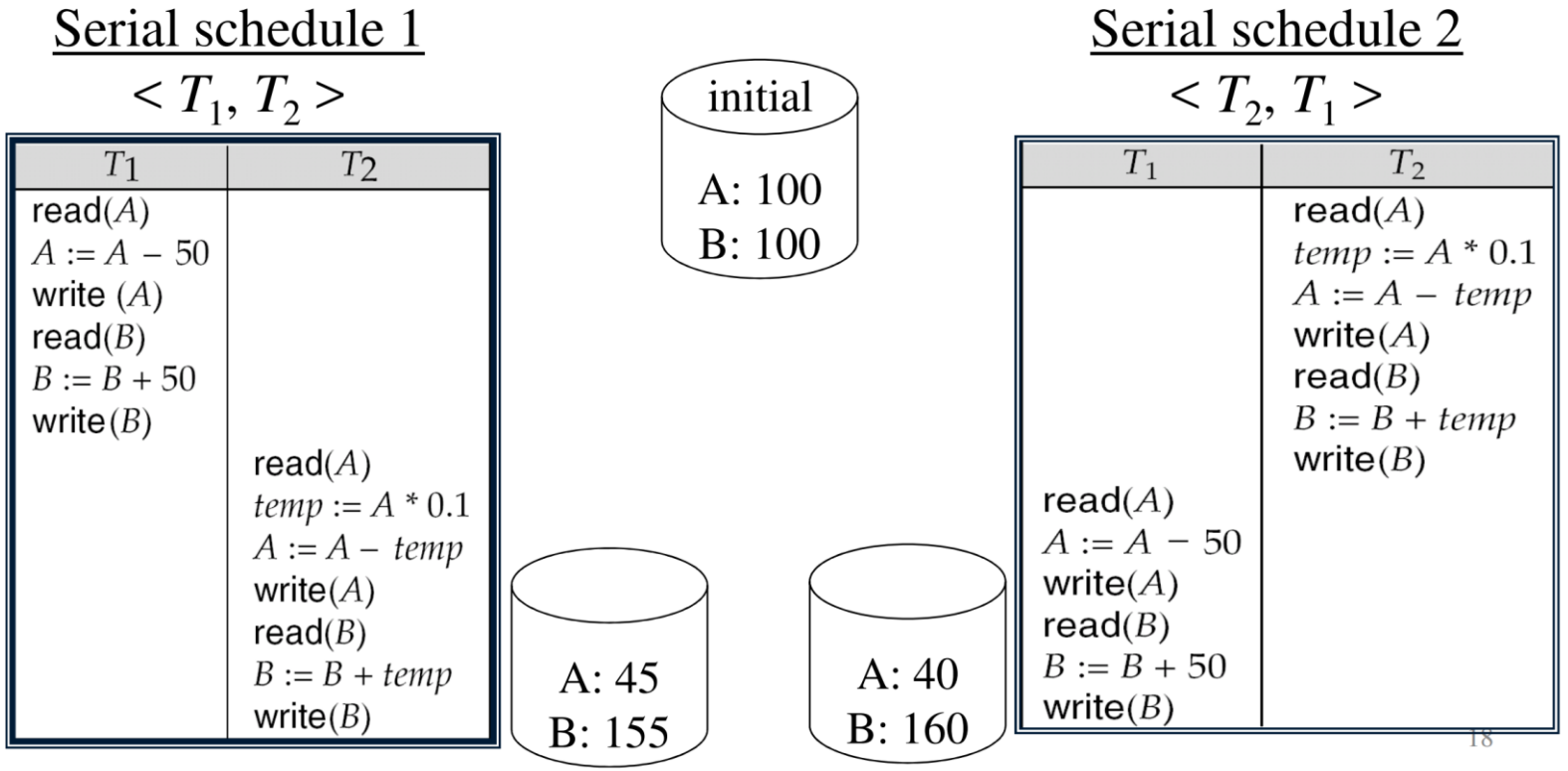
Time

not a schedule!

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

Schedules

- **Serial schedule:** the next transaction begins only after the current one completes
  - Let T1 transfer $50 from A to B,
    and T2 transfer 10% of the balance from A to B



Examples of Concurrent Schedules

Schedule 3

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)	
	read(B)
	$B := B + temp$
	write(B)

Schedule 4

T_1	T_2	A, B
read(A)		100, 100
$A := A - 50$		
	read(A)	
	$temp := A * 0.1$	
	$A := A - temp$	
	write(A)	90, 100
	read(B)	
write(A)		50, 100
read(B)		
$B := B + 50$		
write(B)		
	$B := B + temp$	
	write(B)	

- Schedule 3 is not a serial schedule, but it is **equivalent** to Schedule 1
- The sum A + B is preserved in Schedules 1, 2, 3, but it is **not preserved** in Schedule 4

Serializability

- **Basic Assumption** – Each transaction preserves database consistency
  - Thus any serial schedule preserves database consistency

- A schedule is **serializable** if it is equivalent to a serial schedule. Two forms of schedule equivalence:
  1. **conflict serializability**
  2. **view serializability**

- *Simplified view of transactions*
  - We ignore operations other than **read** and **write**
  - Transactions may perform any computation on data in local buffers (in main memory) in between reads and writes

Conflicting Instructions

	l_i = **read**(Q)	l_i = **write**(Q)
l_j = **read**(Q)	No conflict	Conflict
l_j = **write**(Q)	Conflict	Conflict

- Let l_i and l_j be instructions of transactions T_i and T_i respectively
- l_i and l_j are **conflict** when
  - both of them access the same data item Q_j , and
  - at least one of them is **write**(Q)
- The result remains the same if we swap two consecutive instructions l_i and l_j that do not conflict
- A conflict between l_i and l_j forces a temporal order between them

Conflict Serializability

- Two schedules S and S' are **conflict equivalent** if S can be transformed into S by a series of swapping non-conflicting instructions
- A schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule.

Any conflict between these two instructions?

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

Schedule S

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

A serial schedule

Conflict Serializability (Cont.)

- Is this schedule conflict serializable?

Can we swap them ?

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

Can we swap them ?

- NO because
  - Possible serial schedules are: $< T_3, T_4 >$ or $< T_4, T_3 >$
  - But we are unable to swap instructions in the above schedule to obtain either serial schedule

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 satisfied, for each data item Q,
  1. The transaction (if any) that performs the first **read(Q)** operation in schedule S must also perform the first **read(Q)** operation in schedule S'.
  2. The transaction (if any) that performs the final **write(Q)** operation in schedule S must also perform the final **write(Q)** operation in schedule S'.
  3. If in schedule S transaction T_i reads the value of Q produced by transaction T_j, then in schedule S' also transaction T_i must read the value of Q produced by the same **write(Q)** operation of transaction T_j.

- View equivalence is also based purely on **reads** and **writes** alone.

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.

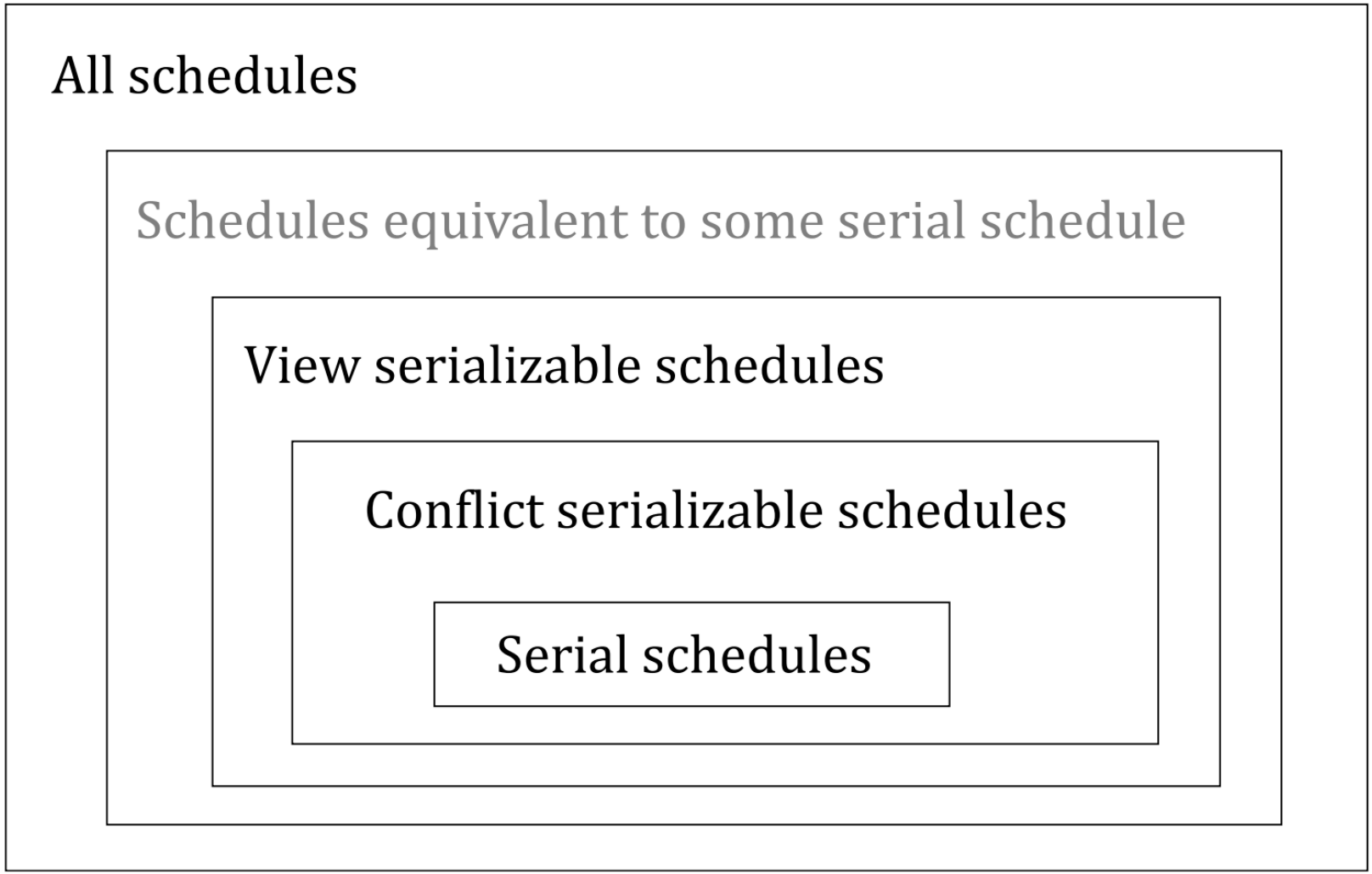
View Serializability (Cont.)

- Every conflict serializable schedule is also view serializable.
- Some schedule (e.g., the one below) is view-serializable but not conflict serializable.
  - Such schedule has **blind write(s)**

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- Which serial schedule is the above equivalent to?
  - $< T_3, T_4, T_6 >$ or $< T_4, T_3, T_6 >$

Serializability of Schedules



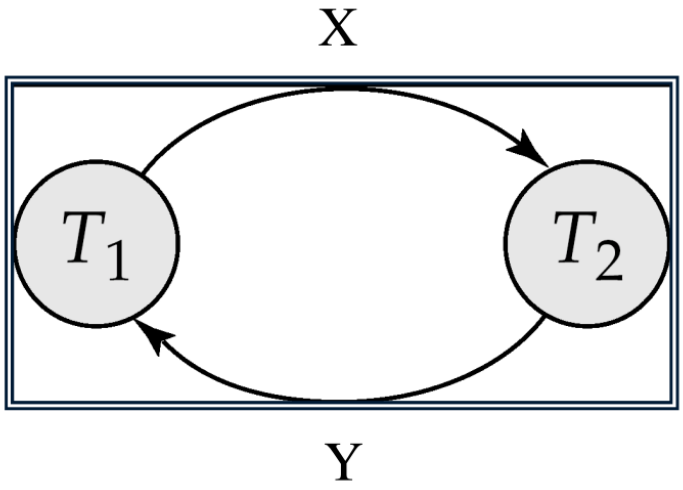
Outline

- ✓ **Basic concepts** on transaction processing
- ✓ What is a serializable schedule?
- ✓ How to test for a serializable schedule?
- ❖ What is a recoverable schedule?

View Serializability (Cont.)

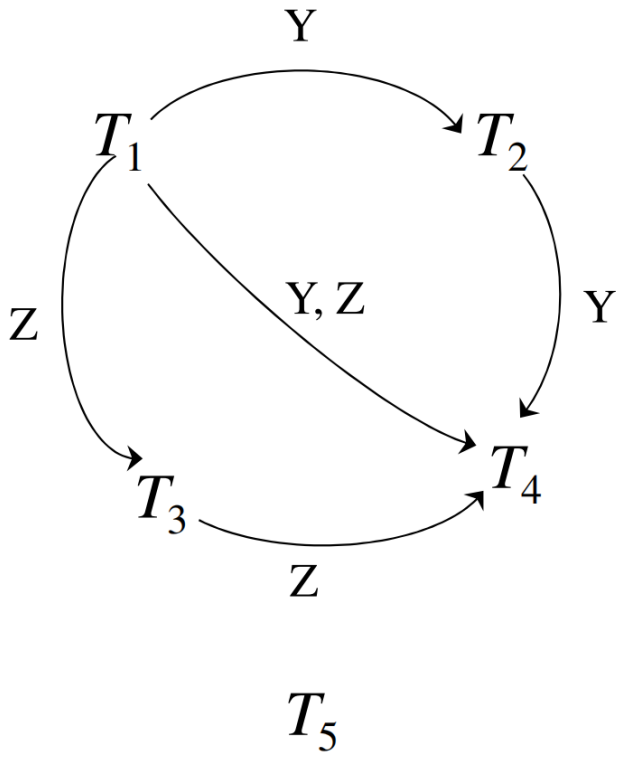
- Consider a schedule of a set of transactions $T_1, T_2, \ldots, T_n$
- **Precedence graph**
  - A *vertex* denotes a transaction (name)
  - A *directed arc* from T_i to T_j if the two transactions conflict on a data item, and T_i accessed that data item earlier
    - Label the arc by the corresponding item

T_1	T_2
write(X)	
	read(X)
	write(Y)
read(Y)	



Example: Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Test for Conflict Serializability

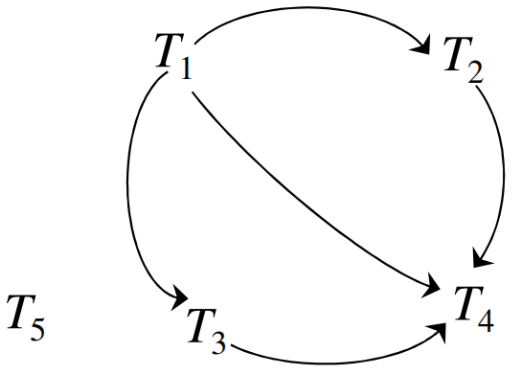
- A schedule is conflict serializable if and only if its precedence graph is **acyclic**

- If precedence graph is acyclic, we can obtain the **serializability order** by a *topological sorting* of the graph
  - Iteratively pick a vertex with no incoming arcs, and then remove all arcs from that vertex
  - Example:

    Consider this precedence graph.

    Find a serializability order for this schedule.

    <___ , ___ , ___ , ___ , ___ >



Test for View Serializability

- The test for view serializability is very expensive
  - E.g., check the schedule against all possible serial schedules
- No known algorithm to solve this problem efficiently
- Some **subclasses** of view serializability can be checked efficiently
  - These algorithms are out of the scope of this course

Outline

- ✓ **Basic concepts** on transaction processing
- ✓ **What is a serializable schedule?**
- ✓ **How to test for a serializable schedule?**
- ✓ **What is a recoverable schedule?**

We must consider the effect of transaction failures on concurrently running transactions.

Recoverable Schedules

- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the **commit** operation of T_i appears before the **commit** operation of T_j
- The following schedule is not recoverable if T_9 commits immediately after the read!

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

- If T_8 may abort in future, T_9 would have read (and shown to the user) an inconsistent database state!
- Hence, DBMS must ensure that schedules are recoverable

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
- Example: the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

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

- If T_{10} fails, then we must also roll back T_{11} and T_{12}
- May undo a large amount of work ☹

Cascadeless Schedules

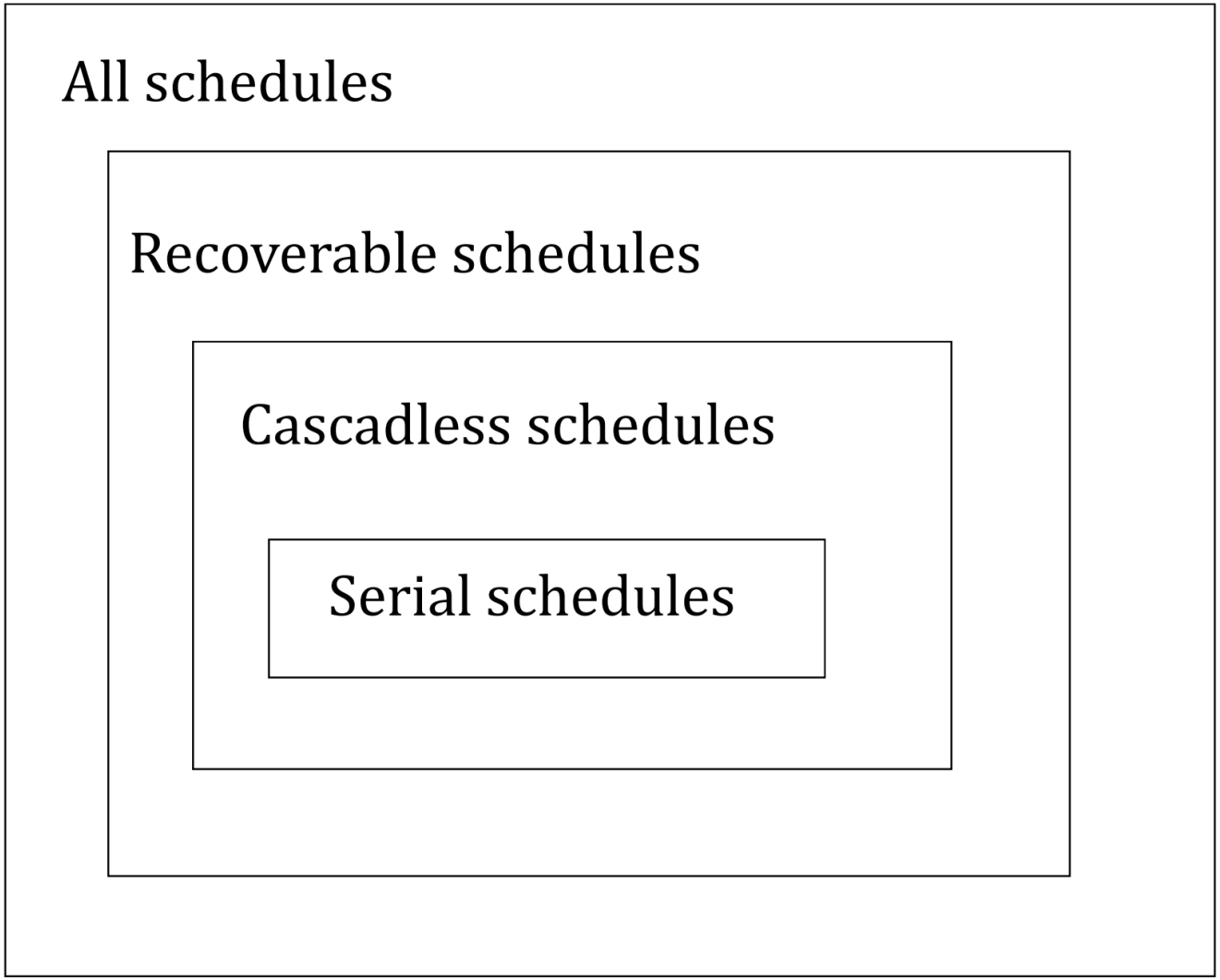
- **Cascadeless schedule** — cascading rollbacks cannot occur;
  - for each pair of transactions T_i and T_j such that

    T_j reads a data item previously written by T_i,

    the commit operation of T_i appears before the read operation of T_j
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to cascadeless ones

T_1	T_2	T_3
read(A)		
read(B)		
write(A)		
commit		
	read(A)	
	commit	
		read(A)
		read(A)
		commit

Recoverability of Schedules



Concurrency Control

- A database must provide a mechanism to ensure that all possible schedules are
  - either conflict or view **serializable**, and
  - are **recoverable** and preferably cascadeless
- A simple policy: execute only one transaction at a time.
  - This generates serial schedules, but provides poor concurrency.
  - Are serial schedules recoverable/cascadeless?
- Too late to test a schedule for serializability after executing it!
- **Goal** – develop concurrency control protocols (e.g., locking) to guarantee that schedules are serializable, recoverable and cascadeless
  - These protocols do not need to examine the precedence graph ☺

Weak Levels of Consistency

- Some applications accept weak levels of consistency and non-serializable schedules
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
    - Supported by setting the transaction isolation level: https://msdn.microsoft.com/en-us/library/ms173763.aspx
- Tradeoff accuracy for performance

Conclusions

- **ACID** properties
- Isolation requirements for concurrent executions (I)
  - ensured by serializable schedules
  - conflict serializability, view serializability
  - [Future] we will study concurrency control schemes which produce serializable schedules
- Transaction failures (A,D)
  - ensured by recoverable and cascadeless schedules
  - [Future] we will study the recovery manager component which supports atomicity and durability

南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

DBGroup

谢谢！

DBGroup @ SUSTech

Dr. Bo Tang (唐博)

tangb3@sustech.edu.cn