Lecture16: Transaction

CS211 - Introduction to Database

Transaction

- Transaction refers to a collection of operations that form a single logical unit of work.
  - ❑ transfer of money from one account to another is a transaction consisting of two updates, one to each account

- For a successful transaction, a database system must ensure the following:
  1. Proper execution of transactions despite failures — **Either the entire transaction executes, or none of it does**.
  2. It must **manage concurrent execution of transactions** in a way that avoids the introduction of data inconsistency.

Transaction Initiation

- Usually, a transaction is initiated by a user program written in a **high-level DML**(typically SQL), or **programming language** (for example, C++, or Java), **with embedded database accesses** in JDBC (Java Database Connectivity) or ODBC (Open Database Connectivity).

Begin transaction

/* all the transaction related operations*/

End transaction

This collection of steps must appear to the user as a single, indivisible unit

Transaction Failure

- A transaction may fail during its execution due to any of these reasons:
    - Divided by zero error
    - Operating system crashed
    - The computer stopped operating
    - Network failure
    - Firewall rejection etc.

- If a transaction begins to execute but fails for whatever reason, any changes to the database that the transaction may have made must be **undone**.

Transaction – ACID properties

- **Atomicity**
  - Either all operations of the transaction are reflected properly in the database, or none are. (**all-or-none**)
- **Consistency**
  - Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
- **Isolation**
  - Each transaction is unaware of other transactions executing concurrently in the system.
  - Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished.
- **Durability**
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

A simple transaction model

- We shall understand the transaction concept using a simple **bank application** consisting of several accounts and a set of transactions that access and update those accounts.

- Transactions access data using two operations:

1. **read(*X*):** transfers the ***data item X*** from the database to a ***variable X***, in a buffer in the main memory belonging to the transaction that executed the read operation.
2. **write(*X*):** transfers the value in the ***variable X*** in the main-memory buffer of the transaction that executed the write to the ***data item X*** in the database.

we ignore the insert & delete operations for time being

Transaction example - Money transfer

Let T_i be a transaction that transfers $50 from account *A* to account *B*.

Accounts *A* and *B* are initially having $1000 and $2000 respectively

$$
\begin{aligned}
T_i:\ & \text{read}(A);\\
& A := A - 50;\\
& \text{write}(A);\\
& \text{read}(B);\\
& B := B + 50;\\
& \text{write}(B).
\end{aligned}
$$

1. **Consistency**:
   - The sum of *A* and *B* be **unchanged** by the execution of the transaction.
   - Ensuring consistency for an individual transaction is the responsibility of the **application programmer** who codes the transaction.

Transaction example-Money transfer

2. Atomicity:

- Because of the failure of transaction, the database may be left in an **inconsistent state**.

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

DBMS must ensure that such inconsistencies are not visible in a database system.

Example:

- ❑ Suppose the values of accounts *A* and *B* are \$1000 and \$2000.
- ❑ Suppose that the failure of transaction happened after the write(*A*) operation but before the write(*B*) operation.
- ❑ In this case, the values of accounts *A* and *B* reflected in the database are \$950 and \$2000. The system destroyed \$50 as a result of this failure.

Transaction example-Money transfer

2. Atomicity:

Ensuring atomicity is the job of DBMS **recovery system.** The basic idea behind ensuring atomicity is this:

- DBMS writes old values of any data on which a transaction performs a **write** to a file called the ***log*** (on the disk).
- If the transaction does not complete its execution, the database system **restores** the old values from the log to make it appear as though the transaction never executed.

Transaction example-Money transfer

3. Durability:

- The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database **persist**, even if there is a system failure after the transaction completes execution.

- The **recovery system** of the database is responsible for **ensuring durability**, in addition to **ensuring atomicity**.

Transaction example-Money transfer

4. Isolation:

If several transactions are executed **concurrently**, their operations may interleave in some undesirable way, resulting in an ***inconsistent state***.

T_i: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Accounts *A* and *B* are initially having $1000 and $2000 respectively

Example:

- The database is temporarily inconsistent while the transaction to transfer funds from *A* to *B* is executing, with the deducted total written to ***A ($950)*** and the increased total yet to be written to ***B ($2000)***.
- If a second concurrently running transaction reads *A* and *B* at this intermediate point and computes ***A+B***, it will observe an inconsistent value **$2950** instead of **$3000.**

Transaction example-Money transfer

4. Isolation :

- A way to avoid the problem of concurrently executing transactions is to **execute transactions serially**—that is, one after the other.

The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order.

- Ensuring the isolation property is the responsibility of a component of the database system called the **concurrency-control system**.

Storage Structure

To ensure the **atomicity** and **durability** properties of a transaction, we store/retrieve data from following types of storage media:

1. **Volatile storage**:
    - Information residing in volatile storage does not usually survive system crashes.
    - Examples of such storage are main memory and cache memory.
    - Access to volatile storage is extremely fast, both because of the speed of the memory access itself, and because it is possible to access any data item in volatile storage directly.

Storage Structure

2. **Nonvolatile storage**:
    - Information residing in nonvolatile storage survives system crashes.
    - Examples of nonvolatile storage include magnetic disk, flash storage, optical media, and magnetic tapes.
    - Nonvolatile storage is slower than volatile storage, particularly for random access.
    - Both secondary and tertiary storage devices, however, are susceptible to failure which may result in loss of information.

Storage Structure

3. **Stable storage**:

- Information residing in stable storage is *never* lost (in theory).
- Although stable storage is practically impossible to obtain, it can be closely approximated by techniques that make data loss extremely unlikely.
- To implement stable storage, we replicate the information in several nonvolatile storage media (usually disk).

❑ For a transaction to be **atomic**, **log records** need to be written to stable storage before any changes are made to the database on disk.

❑ For a transaction to be **durable**, **its changes** need to be written to stable storage.

Transaction Atomicity and Durability

- A transaction may not always complete its execution successfully. Such a transaction is termed **aborted**.
- Once the changes caused by an aborted transaction have been undone, we say that the transaction has been **rolled back**.
- A rollback is done typically by maintaining a **log** with following steps:

1. We record the **identifier of the transaction** performing the modification.
2. We record the **identifier of the data** item being modified.
3. We record both the **old value** (prior to modification) and the **new value** (after modification) of the data item.
4. Only after steps 1, 2, 3 the database is actually **modified**.

Transaction Atomicity and Durability

- A transaction that completes its execution successfully is said to be **committed,** whose updates to the database **must persist** even if there is a system failure.

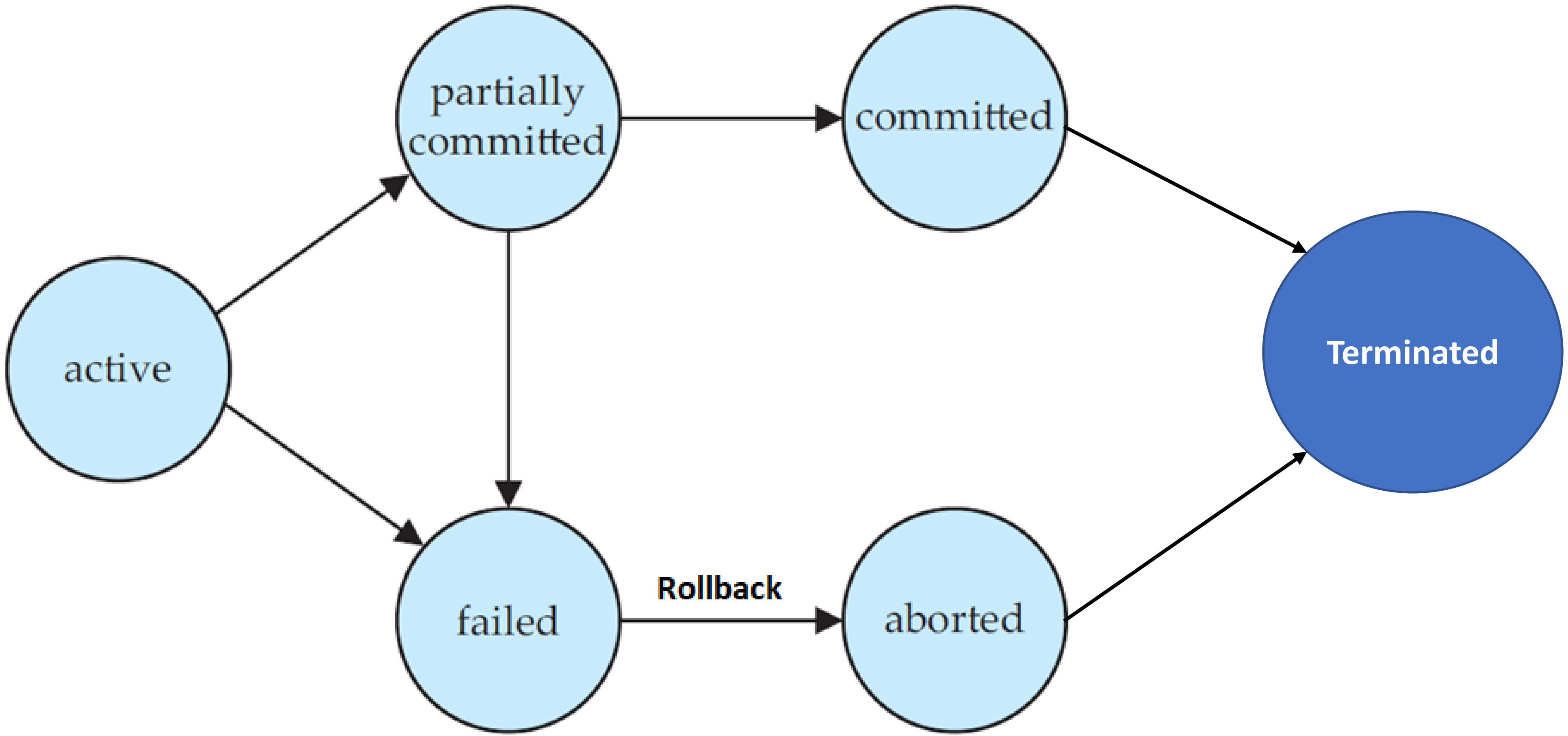
- Once a transaction has committed, **we cannot undo** its effects by aborting it. The only way to undo the effects of a committed transaction is to execute a **compensating transaction**.

- A transaction is said to have **terminated** if it has either **committed** or **aborted**.

- When a transaction finishes its **final statement**, it is **partially committed** and ready to **successfully terminate**.

- A partially committed transaction may **fail** due to hardware failure/software error/ internal logical error.

State diagram of a transaction



When a failed transaction is **rolled back** the system has two options:

1. It can **restart** the transaction if the transaction was aborted as a result of some **hardware or software failure**. A restarted transaction is considered to be a **new transaction**.

2. It can **kill** the transaction if the transaction was aborted because of some **internal logical error**.

Transaction Isolation

- Allowing multiple transactions to update data **concurrently** may cause several complications with consistency of the data.

- **Run transactions Serially**
  - One at a time, each transaction starting only after the previous one has completed

- **Reasons allowing Concurrency**
  - ❑ Improved throughput and resource utilization
    - by concurrently executing I/O & CPU activities
  - ❑ Reduced waiting time
    - reduces average response time

Transaction Isolation

- DBMS maintains the **database consistency while executing concurrent transactions** using **concurrency-control schemes**.

Example:

- The bank accounts *A* and *B* are having $1000 and $2000 respectively
- *The transaction T1* transfers $50 from account *A* to account *B*
- Transaction *T*2 transfers *10* percent of the balance from account *A* to account *B*

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

What is a transaction Schedule ?

- A **schedule** represent the **chronological order** in which **instructions are executed** in the system.

- A schedule for a set of transactions ***must :***

    1. Consist of all instructions of those transactions
    2. Preserve the order in which the instructions appear in each individual transaction

What is a Serial transaction schedule ?

- A **serial schedule** consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

- For a set of ***n*** transactions, there exist *n* factorial **(*n*!)** different valid serial schedules.

Serial schedule-1

A=$1000 and **B=$2000**

	T_1	T_2	
A=1000	read(A)		
A=950	$A := A - 50$		
A=950	write(A)		
B=2000	read(B)		
B=2050	$B := B + 50$		
B=2050	write(B)		
	commit		
		read(A)	A=950
		$temp := A * 0.1$	temp=95
		$A := A - temp$	A=855
		write(A)	A=855
		read(B)	B=2050
		$B := B + temp$	B=2145
		write(B)	B=2145
		commit	

A+B=3000

Database consistency is maintained

Schedule 1—a serial schedule in which ***T1*** is followed by ***T2***

Serial schedule-2 A=$1000 and B=$2000

	T_1	T_2	
		read(A)	A=1000
		$temp := A * 0.1$	temp=100
		$A := A - temp$	A=900
		write(A)	A=900
		read(B)	B=2000
		$B := B + temp$	B=2100
		write(B)	B=2100
		commit	
A=900	read(A)		
A=850	$A := A - 50$		
A=850	write(A)		
B=2100	read(B)		
B=2150	$B := B + 50$		
B=2150	write(B)		
	commit		

A+B=3000 ✓

Database consistency is maintained

Schedule 2—a serial schedule in which ***T2*** is followed by ***T1***

What is a Concurrent transaction schedule ?

- If two transactions are running **concurrently**, the operating system may execute one transaction for a little while, then perform a **context switch**, execute the second transaction for some time, then perform a **context switch** and switch back to the first transaction for some time, and so on.

Concurrent schedule-1

A=$1000 and **B=$2000**

A+B=3000

Database consistency is maintained

	T_1	T_2	
A=1000	read(A)		
A=950	$A := A - 50$		
A=950	write(A)		
		read(A)	**A=950**
	Context Switch	$temp := A * 0.1$	**temp=95**
		$A := A - temp$	**A=855**
		write(A)	**A=855**
		Context Switch	
B=2000	read(B)		
B=2050	$B := B + 50$		
B=2050	write(B)		
	commit		
		read(B)	**B=2050**
	Context Switch	$B := B + temp$	**B=2145**
		write(B)	**B=2145**
		commit	

Schedule 3—a concurrent schedule equivalent to schedule 1

Concurrent schedule-2

A=\$1000 and B=\$2000

	T_1	T_2	
A=1000	read(A)		
A=950	$A := A - 50$		
	Context Switch	read(A)	A=1000
		$temp := A * 0.1$	temp=100
		$A := A - temp$	A=900
		write(A)	A=900
		read(B)	B=2000
A=950	write(A)	Context Switch	
B=2000	read(B)		
B=2050	$B := B + 50$		
B=2050	write(B)		
	commit		
	Context Switch	$B := B + temp$	B=2100
		write(B)	B=2100
		commit	

A+B=3050 ✗

Database consistency is not maintained

Schedule 4—a concurrent schedule resulting in an inconsistent state

Serializability

- A schedule which is **equivalent to a serial schedule** is called a **Serializable schedule.**

A=$1000 and **B=$2000**

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-1 A+B=3000

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

Schedule-3 A+B=3000

Schedule-3 is a concurrent schedule equivalent to serial schedule-1. Hence, it is a Serializable schedule

Serializability

- All the serial schedules are serializable.
- But, if steps of multiple transactions are **interleaved**, it is harder to determine whether a schedule is serializable.

Consider a schedule S in which there are two consecutive instructions, I and J, of transactions Ti and Tj, respectively ($i \neq j$).

1. If I and J refer to **different data items**, then we can swap I and J without affecting the results of any instruction in the schedule.
2. However, if I and J refer to the **same data item** Q, then the order of the two steps may matter.

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

Conflict between transaction instructions in a schedule

- **Two consecutive instructions** I and J **conflict** if they are operations by different transactions T_i and T_j on the **same data item**, and **at least one** of these instructions is a **write** operation.

  - I = read(Q) , J = read(Q) **no conflict**
  - I = read(Q) , J = write(Q) **conflict**
  - I = write(Q) , J = read(Q) **conflict**
  - I = write(Q) , J = write(Q) **conflict**

Schedule-3

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

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

Schedule-3 is a concurrent schedule equivalent to schedule-1

Producing Equivalent Schedule

Let I and J be consecutive instructions of two different transactions T_i and T_j of a schedule ***S***.

- If I and J **do not conflict**, then we can **swap** the order of ***I*** and J to produce a new schedule ***S'***.
- We say ***S' is equivalent to S***.

Producing equivalent schedule - Example

T1	T2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

Swap { T2.write(A) , T1.read(B) }

T1	T2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Producing equivalent schedule - Example

T1	T2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Swap {T2.read(A) , T1.read(B) }

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

Producing equivalent schedule - Example

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
	write(A)
write(B)	
	read(B)
	write(B)

Swap { T2.write(A), T1.write(B) }

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

Producing equivalent schedule - Example

T1	T2
read(A)	
write(A)	
read(B)	
	read(A)
write(B)	
	write(A)
	read(B)
	write(B)

Swap { T2.read(A), T1.write(B) }

T1	T2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule-5 is a serial schedule that is equivalent to **schedule-3**

Not conflict serializable schedule - Example 1

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

Schedule-6 is a not a conflict serializable schedule

This schedule is **non-conflict serializable**, since it is not equivalent to either the serial schedule ***<T3,T4>*** or the serial schedule ***<T4,T3>***.

Precedence graph

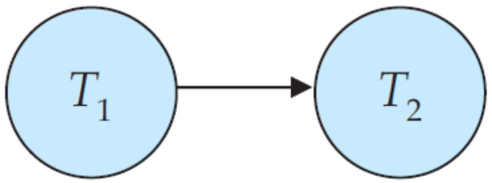
- It is a simple and efficient method for **determining conflict Serializability** of a schedule.

- Consider a schedule *S*. A **precedence graph** is a **directed graph *G* = (*V, E*)** of *S* where*:*
  - ❑ ***V*** is a set of vertices consists of all the transactions participating in the schedule
  - ❑ ***E*** is a set of edges that consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:
    - T_i executes write(Q) before T_j executes read(Q)
    - T_i executes read(Q) before T_j executes write(Q)
    - T_i executes write(Q) before T_j executes write(Q)

Precedence graph for schedule-1 & schedule-2

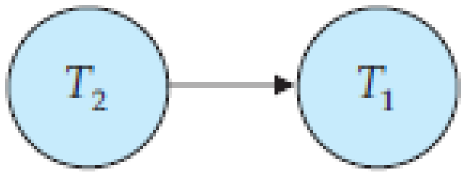
Schedule-1

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

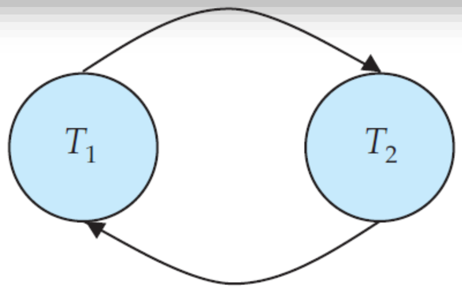
T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	



Precedence graph for schedule-4

Schedule-4

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



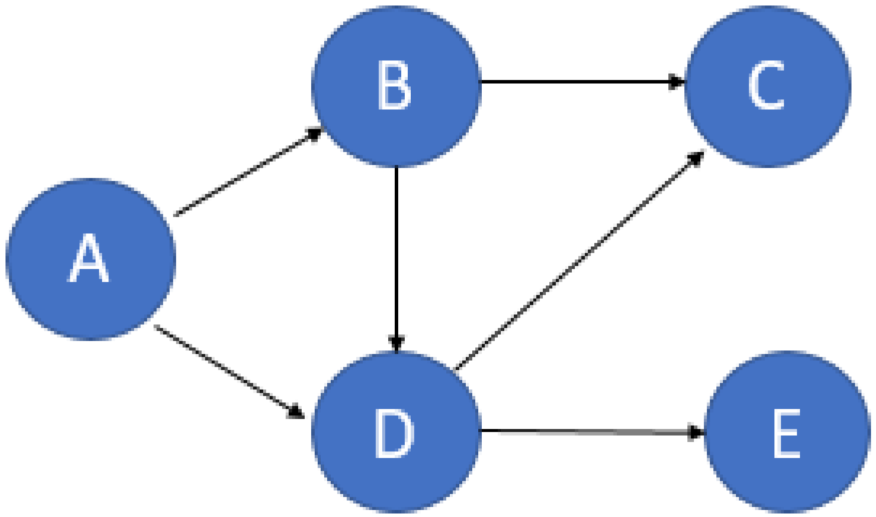
1. Edge *T1* →*T2*, because *T1* executes *read(A)* before *T2* executes *write(A)*.

2. Edge *T2*→*T1*, because *T2* executes *read(B)* before *T1* executes *write(B)*.

Serializability order

A **serializability order** of the transactions can be obtained by finding the **topological sorting (TS)** of the precedence graph.

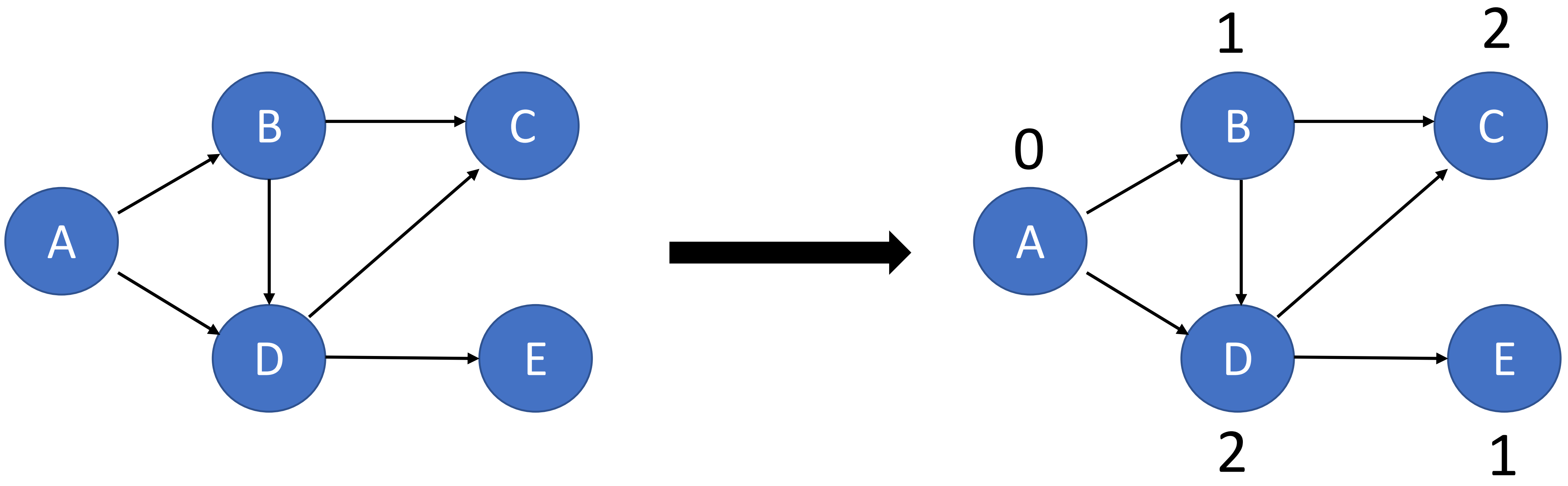
There are, in general, **several possible orders** that can be obtained through a topological sort.



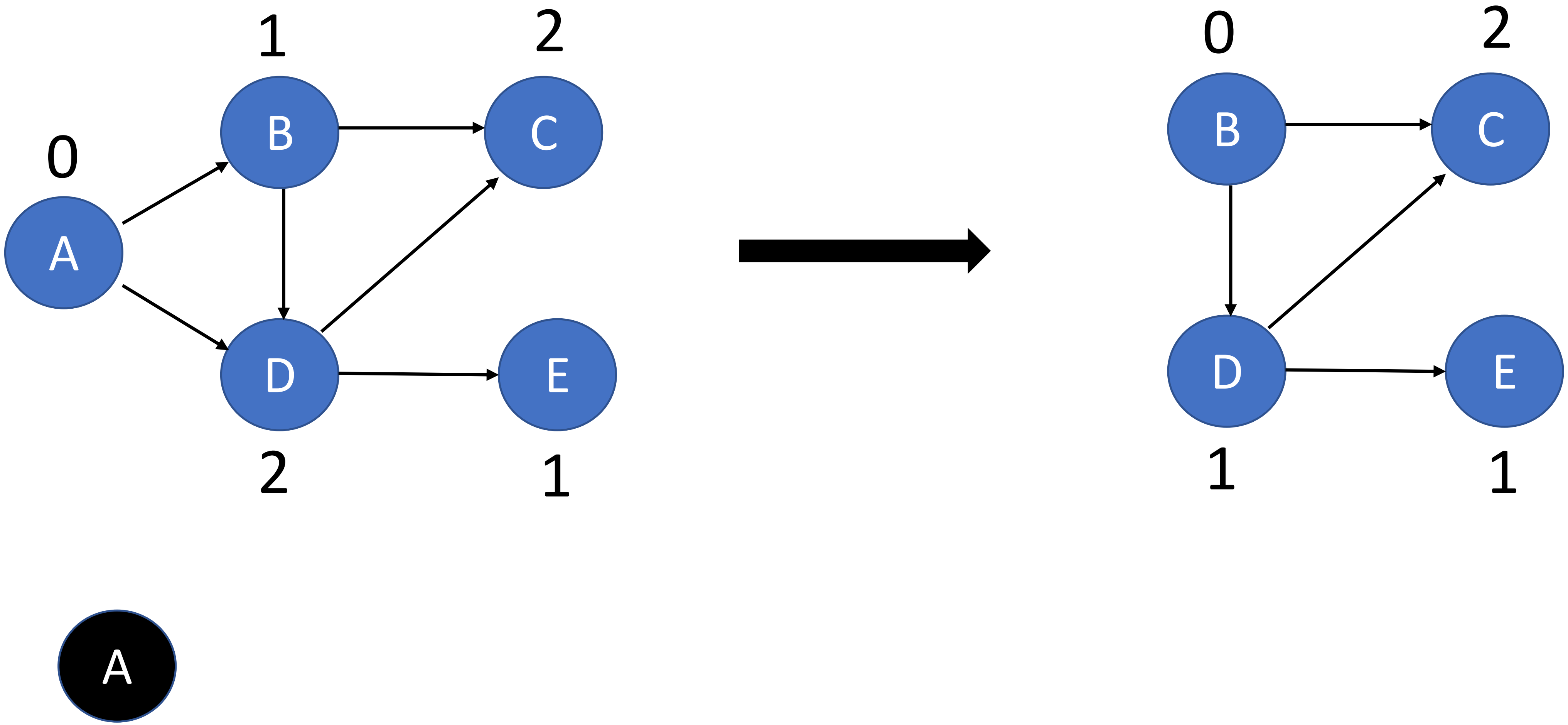
Topological Sorting:

1. List the in-degree of every vertex.
2. Select the vertex with in-degree of zero and mark it as visited. Remove this vertex from the graph.
3. Repeat step-2 until all the vertices are visited in the graph.
4. The order in which we have visited the vertices determines the topological sorting of the precedence graph.

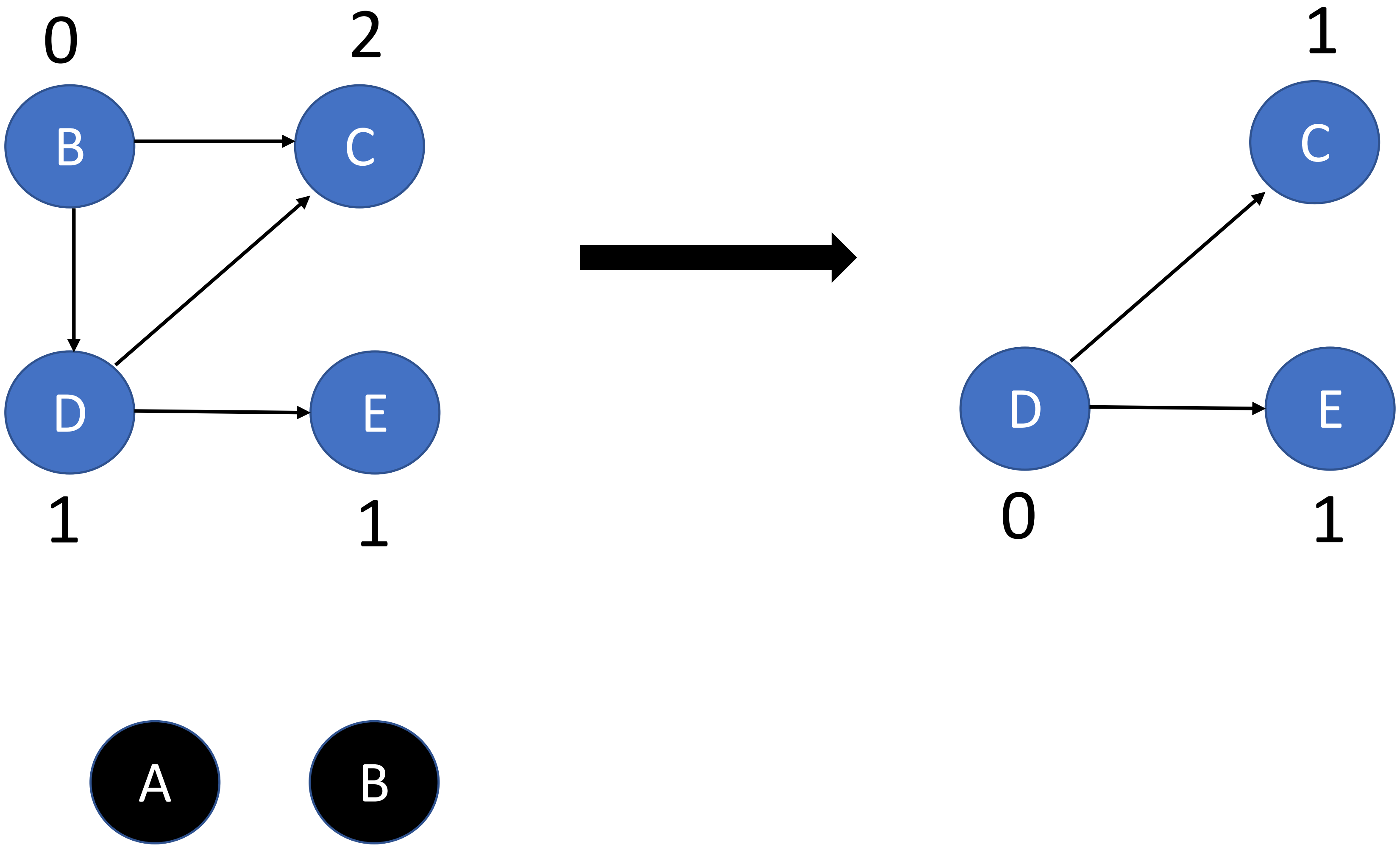
Topological Sorting



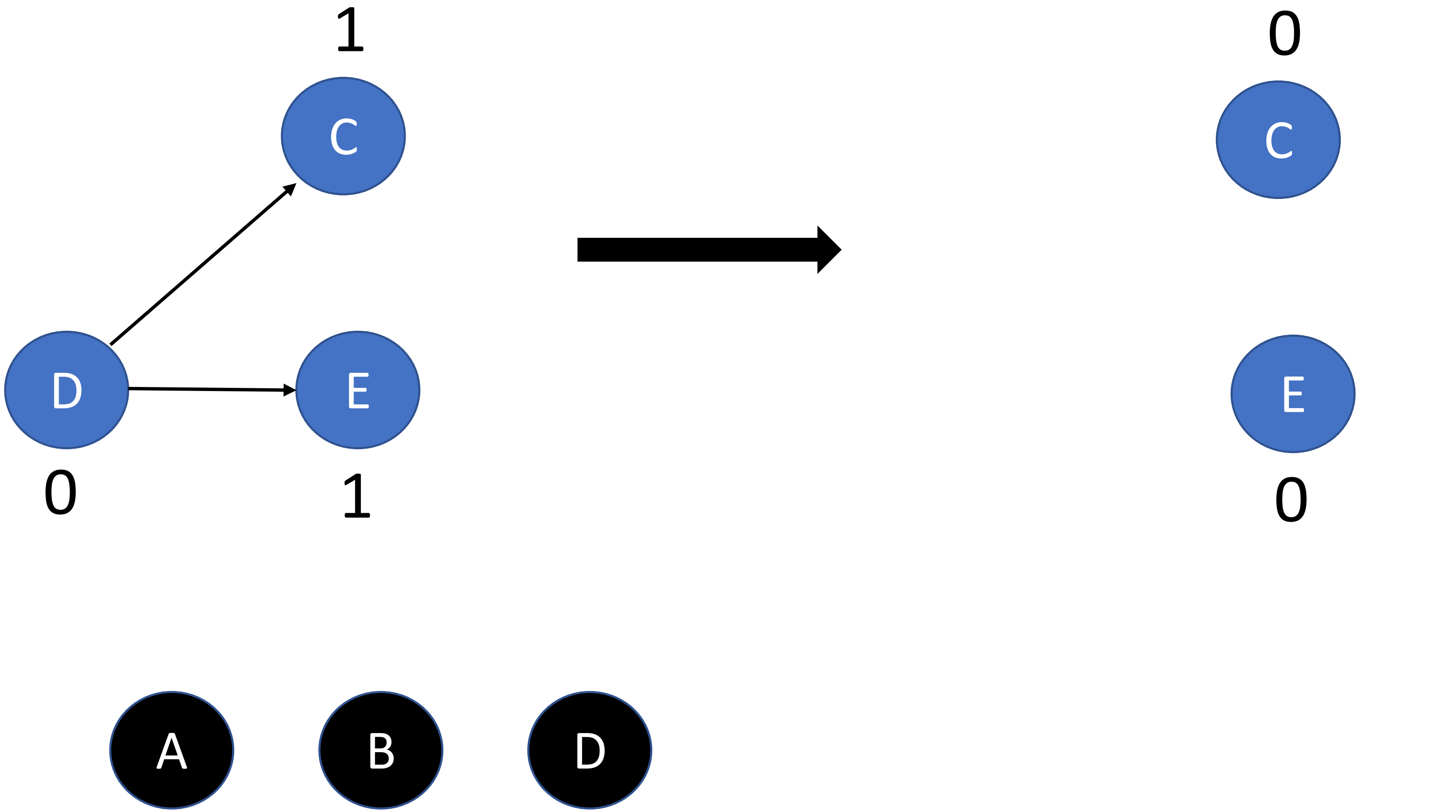
Topological Sorting



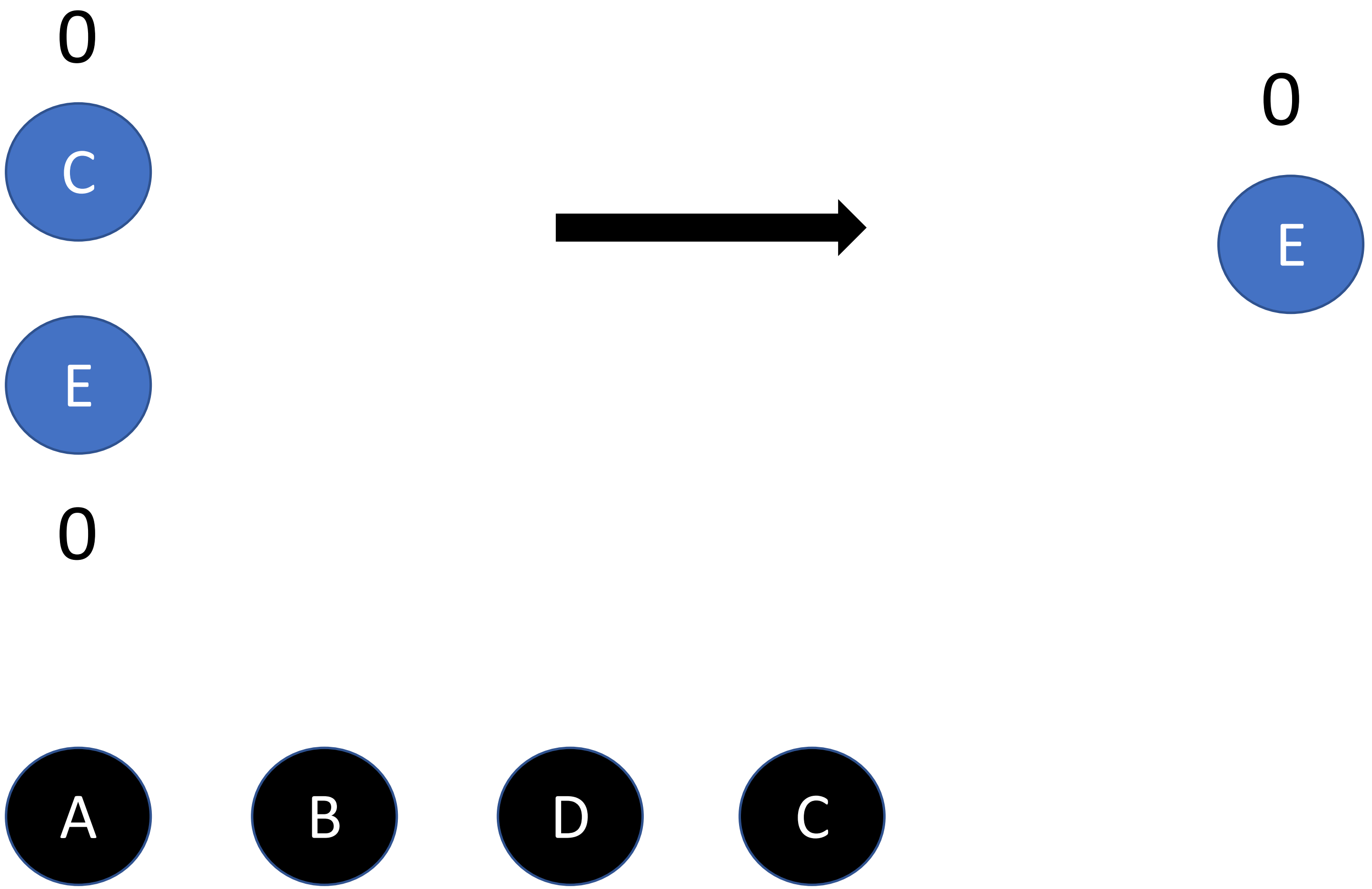
Topological Sorting



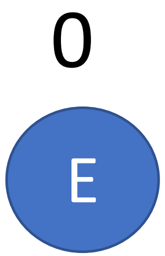
Topological Sorting



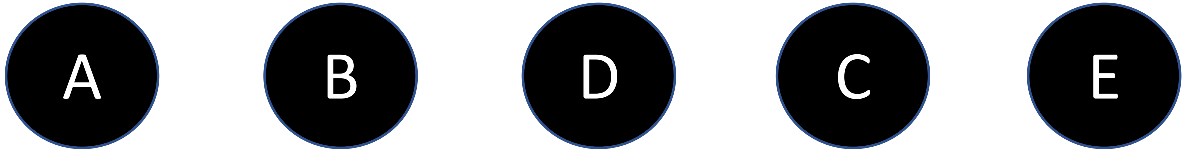
Topological Sorting



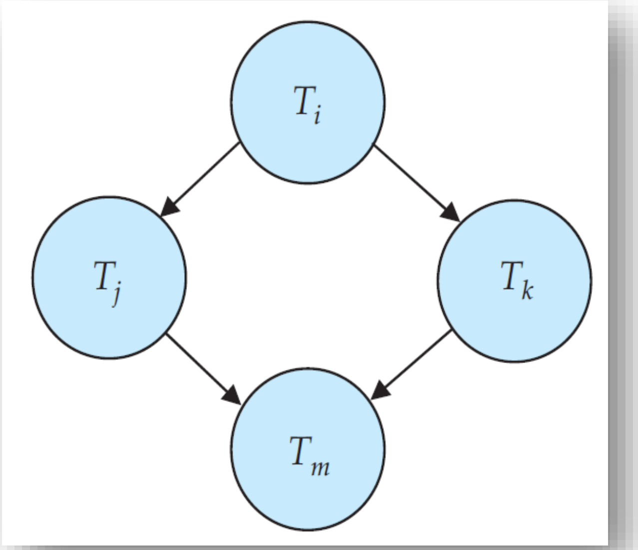
Topological Sorting



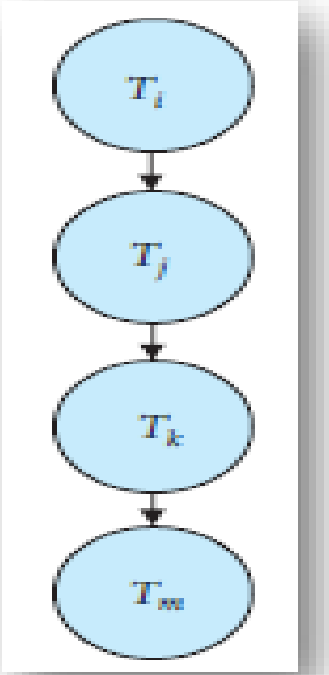
All vertices are visited



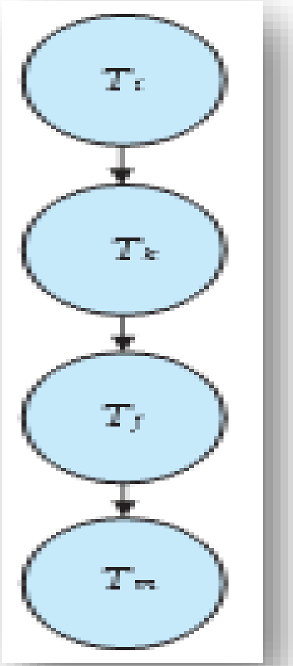
Serializability order example-1



Precedence graph



Topological Sorting -1



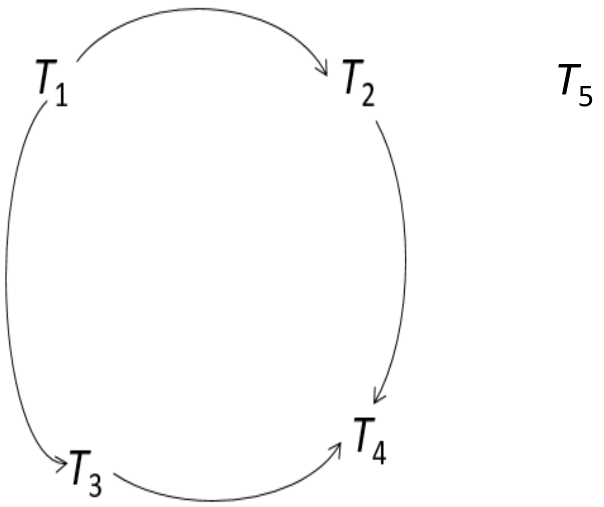
Topological Sorting -2

Serializability order example-2

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)				

Serializability order example-2

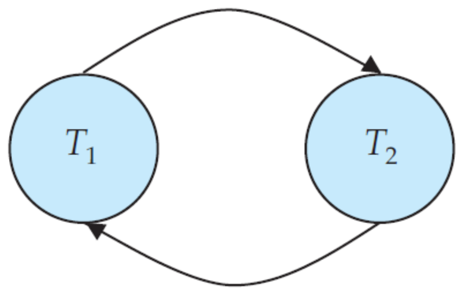
T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



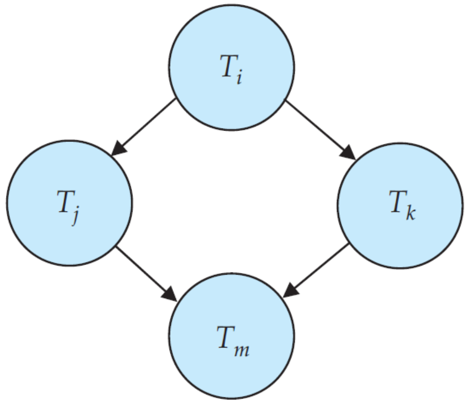
$T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$

Presence of Cycle in precedence graph

If the precedence graph for *S* has a cycle, then schedule *S* is not conflict serializable.



If the graph contains no cycles, then the schedule *S* is conflict serializable.



Not conflict serializable schedule- Example 2

- It is possible to have two schedules that produce the **same outcome**, but that are **not conflict equivalent**.
- In such case, the system must analyse the ***computation performed by transactions***, rather than just the **read** and **write** operations.

A=$1000 and B=$2000

A+B=3000 ✓

	T1	T5	
A=1000	read(A)		
A=950	A:=A-50		
A=950	write(A)		
		read(A)	A=950
		read(B)	B=2000
		B:=B-10	B=1990
		write(B)	B=1990
B=1990	read(B)		
B=2040	B:=B+50		
B=2040	write(B)		
		A:=A+10	A=960
		write(A)	A=960

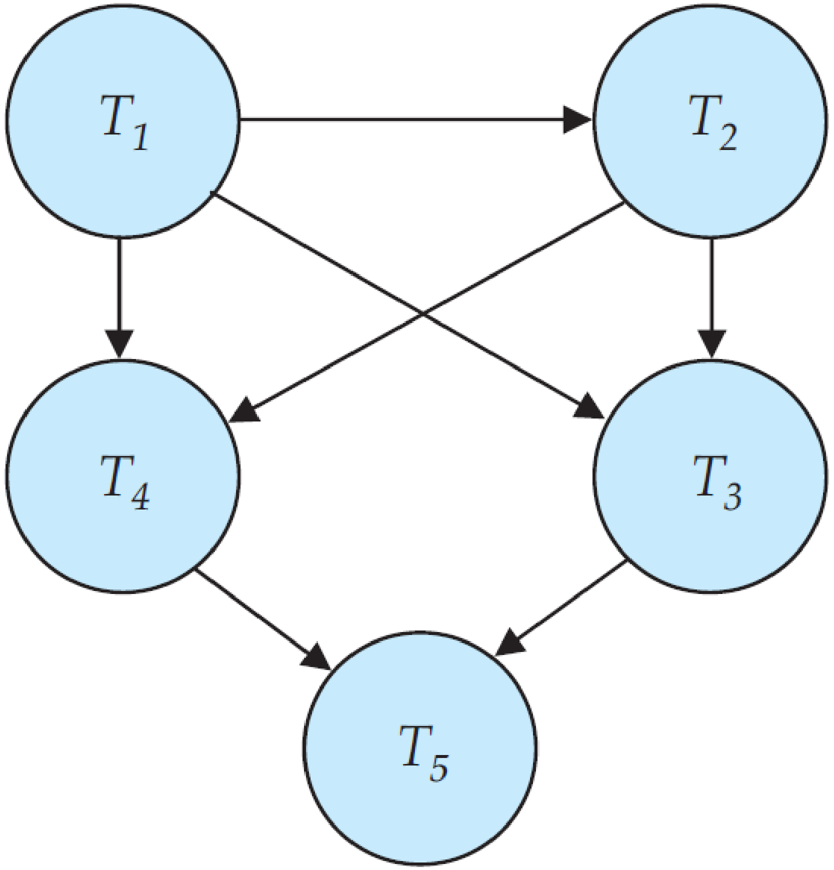
Schedule-6

$$T_1 \rightleftarrows T_5$$

	T1	T5	
A=1000	read(A)		
A=950	A:=A-50		
A=950	write(A)		
B=2000	read(B)		
B=2050	B:=B+50		
B=2050	write(B)		
		read(B)	B=2050
		B:=B-10	B=2040
		write(B)	B=2040
		read(A)	A=950
		A:=A+10	A=960
		write(A)	A=960

Schedule-7

Is the corresponding schedule conflict serializable?



Important definitions in transaction management:

- A **transaction** is a unit of program execution that accesses and possibly updates various data items.

- **Throughput** of the system is, the number of transactions executed in a given amount of time.

- **Average response time** is the average time for a transaction to be completed after it has been submitted.

- **Schedules** represent the chronological order in which instructions are executed in the system.

- A **serial schedule** consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.

Important definitions in transaction management:

- We say that two instructions *I* and *J* **conflict** if they are operations by different transactions on the same data item, and at least one of these instructions is a write operation.

- 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.