Lecture 9

Transaction Management

Dr. Vandana Kushwaha

Department of Computer Science

Institute of Science, BHU, Varanasi

Introduction

- **Collections of operations** that form a **single logical unit** of **database processing** are called **Transactions.**
- A **transaction** includes one or more **database access operations**—these can include **insertion, deletion, modification**, or **retrieval operations.**
- If the **database operations** in a transaction **do not update** the **database** but **only retrieve data**, the **transaction is called a read-only transaction**; otherwise it is known as a **read-write transaction.**
- To **ensure integrity** of the data, we require that the **database system** maintain the following **properties of the transactions**:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

Properties of Transaction

- **ATOMICITY**
- Either **all operations** of the **transaction** are **reflected properly** in the **database**, or **none are.**
- **CONSISTENCY**
- Execution of a **transaction** in **isolation** (that is, with no other transaction executing concurrently) **preserves the consistency** of the **database.**
- **ISOLATION**
- Even though **multiple transactions** may **execute concurrently**, the system guarantees that, for every **pair of transactions** ***Ti*** and ***Tj*** ,
  - It appears to ***Ti*** that either ***Tj*** **finished execution** before ***Ti*** **started**, or
  - ***Tj*** **started execution** after ***Ti*** **finished.**
  - Thus, **each transaction** is unaware of other **transactions executing concurrently** in the system.

Properties of Transaction

- **DURABILITY**
- After a **transaction completes successfully**, the **changes it has made** to the **database persist**, even if there are **system failures.**

- **Note:** These **properties** of **transactions** are also known as **ACID properties** of transaction.

Properties of Transaction: Example

- Consider a simplified **banking system** consisting of **several accounts** and a set of **transactions** that **access** and **update** those **accounts.**
- We assume that the **database permanently resides on disk**, but that **some portion** of it is temporarily residing in **main memory buffer.**
- **A Transactions** can access **data** using **two operations:**
  - **read(*X*),** which **transfers the data item *X*** from the **database to a local buffer** belonging to the transaction that executed the **read operation.**
  - **write(*X*),** which **transfers the data item *X* from the local buffer** of the transaction that executed the **write back to the database.**

Properties of Transaction: Example

- Let ***Ti*** be a **Transaction** that **transfers $50** from **account *A*** to **account *B*.**
- This **transaction** can be defined as:
  - ***Ti*: read(*A*);**
  - ***A*** := *A* − 50;
  - **write(*A*);**
  - **read(*B*);**
  - ***B*** := *B* + 50;
  - **write(*B*).**

Properties of Transaction: Example

- **ATOMICITY**
- Suppose that, just **before the execution** of **transaction *Ti*** the values of **accounts *A*** and ***B*** are **$1000 and $2000**, respectively.
- Now suppose that, during the execution of **transaction *Ti*,** a **failure occurs** that prevents ***Ti*** from completing its execution successfully.
- Examples of such failures include **power failures, hardware failures, and software errors.**
- Further, suppose that the **failure 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.**

Properties of Transaction: Example

- In particular, we note that the sum ***A* + *B* is no longer preserved.**
- The basic idea behind **ensuring atomicity** is this: The **database system keeps track (on disk) of the old values** of any **data** on which a transaction performs a **write.**
- And, if the **transaction does not complete its execution**, the **database system restores the old values** to make it **appear as** though the **transaction never executed.**
- Ensuring **Atomicity** is handled by a component called the **Transaction-management component** of **DBMS** software .

Properties of Transaction: Example

- **CONSISTENCY**
- The **Consistency** requirement here is that the **sum of *A* and *B*** be **unchanged** by the **execution** of the **transaction.**
- Without the **consistency requirement**, **money** could be **created or destroyed** by the **transaction!**
- Ensuring **consistency** for an individual transaction is the **responsibility** of the **Application programmer** who **codes the transaction**.

Properties of Transaction: Example

- **ISOLATION**
- If **several transactions** are **executed concurrently**, their operations may **interleave** in some **undesirable way**, resulting in an **inconsistent state.**
- For **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* and the increased total yet to be written to *B*.**
- If a **second concurrently running transaction reads *A* and *B*** at this **intermediate point** and computes ***A+B*,** it will observe an **inconsistent value.**
- Furthermore, if this **second transaction** then performs updates on ***A*** and ***B*** based on the **inconsistent values** that it read, the **database** may be left in an **inconsistent state** even after both transactions have completed.

Properties of Transaction: Example

T1	T2
read(A)	
A=A-50	
write(A)	
Read(B)	
B=B+50	
Write(B)	
	read(A)
	temp= A*0.1
	A= A – temp
	write(A)
	Read(B)
	B=B+temp
	write(B)

T1	T2
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)	
	B=B+temp
	write(B)

Properties of Transaction: Example

- A way to avoid the problem of concurrently executing transactions is to execute **transactions serially**—that is, one after the other.
- However, **concurrent execution of transactions provides significant performance benefits.**
- 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 component.**

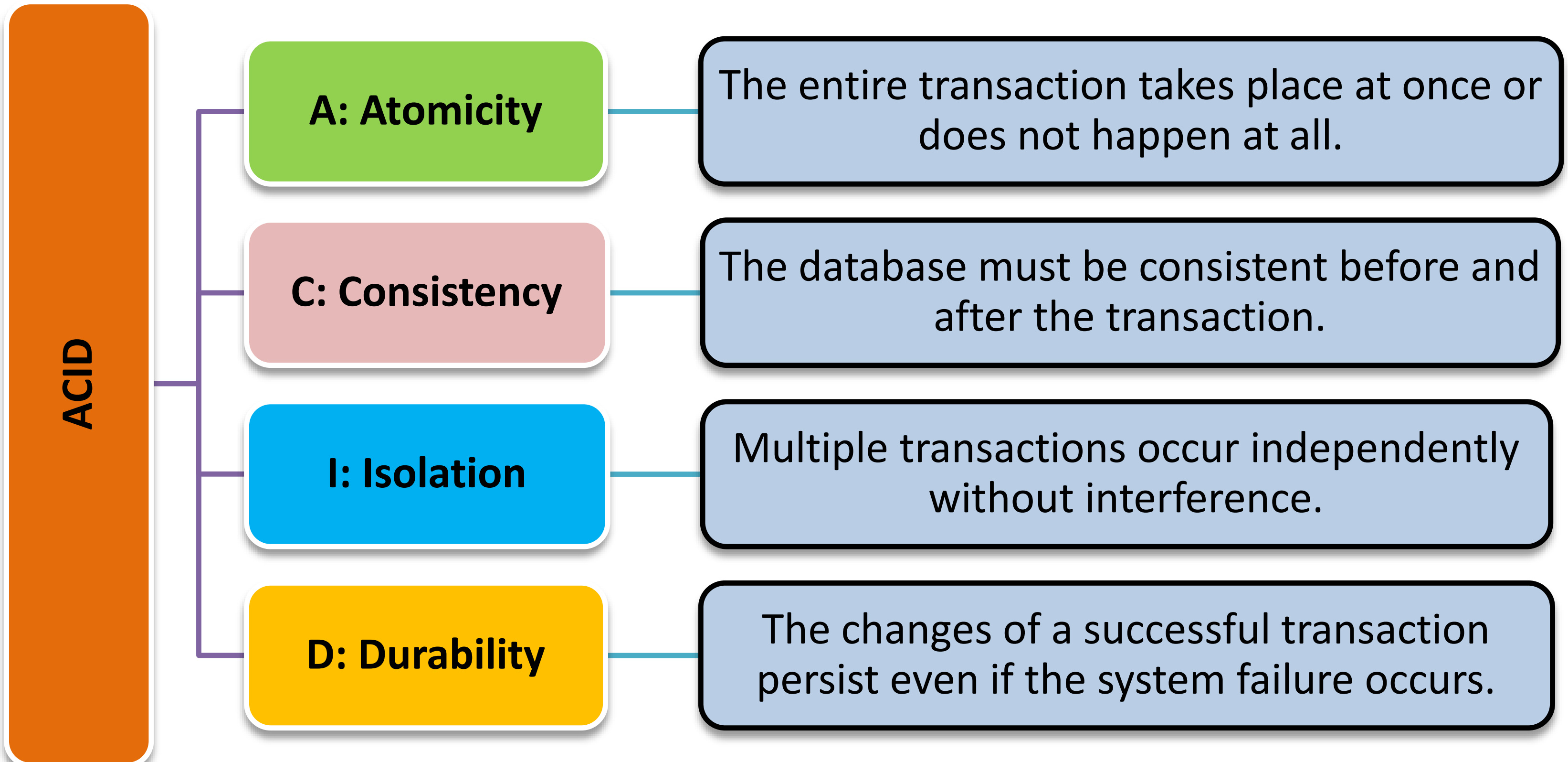
Properties of Transaction: Example

- **DURABILITY**
- Once the **execution** of the **transaction completes successfully**, and the user who initiated the transaction has been notified that the transfer of funds has taken place.
- It must be the case that **no system failure** will **result in a loss of data** corresponding to this **transfer of funds.**
- 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.

Properties of Transaction: Example

- We assume that a **failure of the computer system** may result in **loss of data** in **main memory**, but **data** written to **disk** are **never lost.**
- We can guarantee **Durability** by ensuring that either:
  - The **updates carried out by the transaction** have been **written to disk** before the **transaction completes.**
  - **Information about** the **updates** carried out by the **transaction** and written to **disk** is sufficient to enable the **database** to **reconstruct the updates** when the **database system is restarted after** the **failure.**
- Ensuring **Durabilit**y is the **responsibility** of a **component** of the database system called the **Recovery-management component.**

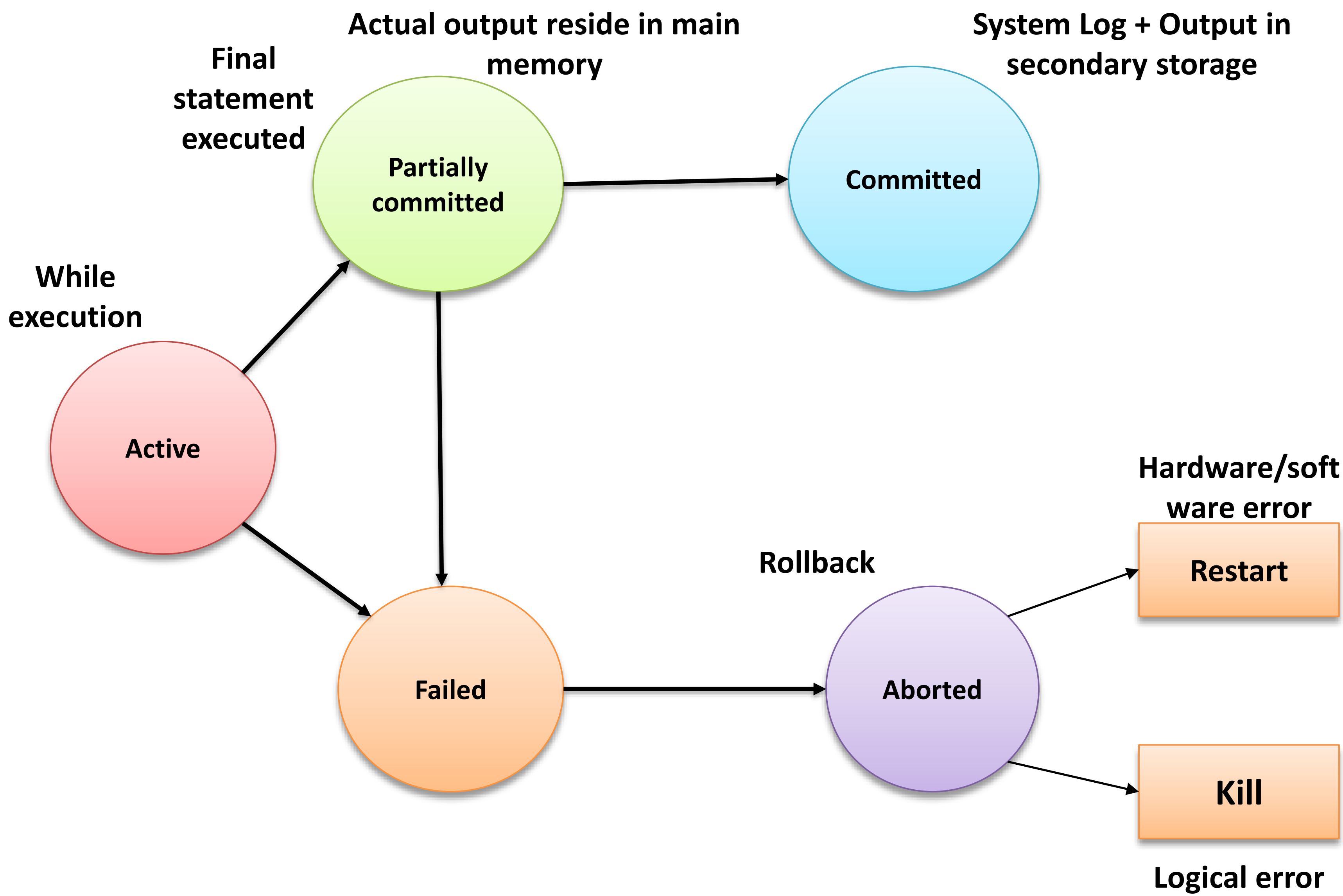
Properties of Transaction: Summary



Transaction States

- A **Transaction** must be in one of the following **states:**
- **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 has been **restored** to its **state** prior to the start of the transaction.
- **Committed**
  - After **successful completion**.

Transaction States



Transaction States

- A **Transaction starts** in the **Active state.**
- When it **finishes** its **final statement**, it enters the **Partially committed state.**
  - At this point, the **transaction** has **completed** its **execution**,
  - But it is **still possible** that it may have to be **aborted**, since the **actual output** may still be **temporarily residing** in **main memory**, and
  - Thus a **hardware failure** may **preclude** its **successful completion**.
- The **database system** then **writes out** enough information to **disk(log file)** that, even in the event of a **failure**, the **updates** performed by the **transaction** can be **re-created** when the **system restarts** after the **failure.**
- When **the last of this information is written out**, the **transaction enters** the **Committed state.**

Transaction States

- A **Transaction** enters the **Failed state** after the system determines that the **transaction** can **no longer proceed** with its **normal execution** (for example, because of **hardware** or **logical errors**).
- Such a **transaction** must be **Rolled back**, then, it enters the **Aborted state.** At this point, the system has **two options:**
- **Restart the transaction**
  - If the **transaction** was **aborted** as a result of **some hardware or software error**.
  - A **restarted transaction** is considered to be a **new transaction**.
- **Kill the transaction**
  - It usually does so because of some **internal logical error** that can be corrected only by **rewriting** the **application program**, or because the **input was bad**, or because the **desired data** were **not found** in the **database**.

The System Log

- To be able to **recover from failures** that affect **transactions,** the system maintains a **log** to **keep track of all transaction operations** that affect the **values** of **database items.**
- The **log is a sequential, append-only file that is kept on disk**, so it is **not affected** by any type of **failure** except for **disk** or **catastrophic failure**.
- Typically, one (or more) **main memory buffers** hold the **last part of the log file**, so that **log entries** are **first added** to the **main memory** buffer.
- When the **log buffer** is filled, or when certain other conditions occur, the **log buffer** is ***appended*** *to the end of the* ***log file*** *on* ***disk*****.**
- In addition, the **log file** from **disk** is **periodically backed** up to **archival storage (tape)** to guard against **catastrophic failures.**
- The following are the **types of entries**—called **log records**—that are **written** to the **log** file and the corresponding action for each log record:

The System Log

1. **[start_transaction, *T*].**

   Indicates that transaction ***T*** has started execution.

2. **[write_item, *T*, *X*, *old_value*, *new_value*]**.

   Indicates that transaction ***T*** has **changed** the value of database item ***X*** from ***old_value*** to ***new_value.***

3. **[read_item, *T*, *X*]**.

   Indicates that transaction ***T*** has **read** the value of database item ***X*.**

4. **[commit, *T*]**.

   Indicates that transaction ***T*** has **completed successfully**, and affirms that its effect can be committed (recorded permanently) to the database.

5. **[abort, *T*]**.

   Indicates that transaction ***T*** has been **aborted.**

Commit Point of a Transaction

- A **transaction *T*** reaches its **Commit point** when:
  - **All its operations** that access the database **have been executed successfully** *and*
  - the **effect of all the transaction operations on the database** have been **recorded in the log file.**
- **Beyond the Commit point**, the **Transaction** is said to be **Committed**, and its effect must be ***permanently recorded* in the database**.
- The **transaction** then **writes a commit record [commit, *T*]** into the **log.**

Commit Point of a Transaction

- If a **system failure occurs**:
  - We can **search back in the log** for all **transactions** ***T*** that have written a **[start_transaction, *T*]** record into the **log** but have **not written** their **[commit, *T*]** record yet;
  - These **transactions** may have to be ***rolled back* to *undo their effect*** on the **database** during the **recovery process.**
  - **The Transactions** that have **written** their **Commit record [commit, *T*]** in the **log** must also have **recorded** all their **WRITE operations in the log**, so their **effect** on the **database** can be **redone** from the **log records.**

Serial Execution

- The execution sequences of transactions are called **schedules**, shown below:

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

Schedule 1

A serial schedule in which T2 is followed by T1.

- The final values of accounts A and B, after the execution takes place, are $855 and $2145, respectively.
- Thus, the **total amount of money** in **accounts A** and **B**—that is, the **sum A + B** is **preserved** after the execution of both **transactions.**

Serial Execution

- A serial schedule in which T1 is followed by T2.

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

Schedule 2

A serial schedule in which T1 is followed by T2.

- Check the **sum *A* + *B*** is **preserved** or not?
- These schedules(Schedule 1 and Schedule 2) are **serial.**
- Each **serial schedule** consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.
- Thus, for a **set of n transactions**, there exist **n! different valid serial schedules.**

Concurrent Executions

- **Transactions** submitted by the various users may **execute concurrently** and may **access** and **update** the **same database items.**
- If this **concurrent execution is *uncontrolled***, it may lead to **problems**, such as an **inconsistent database.**
- **Example:** Consider two **transactions T1 and T2:**

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

T_2
read_item(X);
$X := X + M$;
write_item(X);

Problems with Concurrent Executions

- **The Lost Update Problem**
- This **problem** occurs when **two transactions** that access the **same database items** have their **operations interleaved** in a way that makes the value of some database items **incorrect.**
- Suppose that **transactions *T1*** and ***T2*** are **submitted** at approximately the **same time**, and suppose that their operations are **interleaved** as shown in Figure:

Time	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);	

Item X has an incorrect value because its update by T_1 is *lost* (overwritten).

- if X = 80 at the start, N = 5 and M = 4 the final result should be X = 79.
- However, in the interleaving of operations it is X = 84.
- Because the update in $T1$ that deducted five from X was ***lost*.**

Temporary Update/Dirty Read Problem

- This **problem occurs** when one **transaction** updates a **database** item and then the **transaction fails** for some reason.
- Meanwhile, the **updated item** is **accessed (read)** by **another transaction** before it is changed back to its **original value**.
- Figure shows an **example** where ***T*1 updates item *X*** and then **fails before completion**, so the **system** must change ***X*** back to its **original value.**
- Before it can do so, however, **transaction *T*2 reads** the ***temporary*** **value** of ***X***, which will not be recorded permanently in the database because of the failure of ***T*1.**
- The value of **item *X*** that is read by ***T*2** is called ***dirty data*** because it has been created by a transaction that has not completed and **committed** yet; hence, this problem is also known as the ***dirty read problem*.**

Temporary Update/Dirty Read Problem

Time	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);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the *temporary* incorrect value of X.

The Incorrect Summary Problem

- If one **transaction** is calculating an **aggregate summary** function on a number of **database items.**
- While other **transactions** are updating some of these items, the **aggregate function** may calculate some values before they are updated and others after they are updated.

T_1	T_3
	$sum := 0$; read_item(A); $sum := sum + A$; ⋮

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

	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;

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

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

Concurrent Execution

- 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, and then switch back to the first transaction for some time, and so on.
- With **multiple transactions**, the **CPU time** is **shared** among all the **transactions.**
- Several **execution sequences are possible**, since the various instructions from both transactions may now be **interleaved.**
- Thus, the **number of possible schedules** for a set of **n transactions** is **much larger than n!.**
- **Example:** Let **two transactions** are executed **concurrently.**
- One possible schedule is **Schedule 3** shown in figure:

Concurrent Execution

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 3—a **concurrent schedule** equivalent to **Schedule 1**.

- After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the **order *T*1 followed by *T*2**.
- The **sum *A* + *B*** is indeed **preserved.**

Concurrent Execution

- **Not all concurrent executions result in a correct state**.
- Consider the schedule of Figure given below:

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

Schedule 4—a **concurrent schedule**.

- After the execution of this schedule, we arrive at a state where the final values of accounts ***A*** and ***B*** are **$950** and **$2100**, respectively.
- This final state is an ***inconsistent state***, since we have **gained $50** in the process of the **concurrent execution.**
- Indeed, the **sum *A* + *B* is not preserved** by the execution of the two transactions.

Concurrent Execution

- We can ensure **consistency of the database** under **concurrent execution** by making sure that **any concurrent schedule** that executed has the **same effect** as a **schedule** that could have **occurred without any concurrent execution.**
- That is, *the **concurrent schedule** should, in some sense, be **equivalent to a serial schedule**.*
- **Serializable schedule**
- A **schedule S** of **n transactions** is **serializable** if it is **equivalent** to some **serial schedule** of the **same n transactions.**
- There are **two types** of **serializability:**

  1. **Conflict Serializability**
  2. **View Serializability**

Conflict Serializability

- Let us consider a **schedule *S*** in which there are **two consecutive** instructions ***Ii*** and ***Ij***, of transactions ***Ti*** and ***Tj*** , respectively.

Schedule S	
Ti	Tj
Ii	
	Ij

- If ***Ii*** and ***Ij*** refer to **different data items**, then we can **swap *Ii* and *Ij*** without affecting the results of any instruction in the schedule. **Example:**

Schedule S	
Ti	Tj
Read(Q)	
	Write(R)

Schedule S	
Ti	Tj
	Write(R)
Read(Q)	

Conflict Serializability

- However, if ***Ii* and *Ij* refer** to the **same data item *Q*,** then the **order** of the two steps may **matter.**
- There are **four cases** that we need to consider:
- **Case 1: Ii = read(Q), Ij = read(Q).**
  - The order of **Ii** and **Ij** does **not matter.**
  - Since the same value of **Q** is read by **Ti** and **Tj** , regardless of the order.

Schedule S	
Ti	**Tj**
Read(Q)	
	Read(Q)

→

Schedule S	
Ti	**Tj**
	Read(Q)
Read(Q)	

Conflict Serializability

- **Case 2: Ii = read(Q), Ij = write(Q).**
  - If **Ii** comes before **Ij,** then **Ti** does not read the value of **Q** that is written by **Tj** in instruction **Ij.**

Schedule S	
Ti	**Tj**
Read(Q)	
	Write(Q)

Schedule S	
Ti	**Tj**
	Write(Q)
Read(Q)	

  - If **Ij** comes before **Ii**, then **Ti** reads the value of **Q** that is written by **Tj.**
  - Thus, the **order of Ii and Ij matters.**
- **Case 3: *Ii* = write(*Q*), *Ij* = read(*Q*).**
  - The order of ***Ii*** and ***Ij*** matters for reasons similar to those of the **previous case.**

Conflict Serializability

- **Case 4: Ii = write(Q), Ij = write(Q).**
  - Since both instructions are **write** operations, the **order** of these instructions **does not affect** either ***Ti* or *Tj*** .

Schedule S

Ti	Tj
Write(Q)	
	Write(Q)
	Read(Q)

Schedule S

Ti	Tj
	Write(Q)
Write(Q)	
	Read(Q)

  - However, the value obtained by the next **read(*Q*)** instruction of ***S*** is affected, since the result of only the **latter** of the two **write instructions** is **preserved** in the database.
  - If there is no other **write(*Q*)** instruction after ***Ii*** and ***Ij*** in ***S***, then the order of ***Ii*** and ***Ij*** directly affects the final value of ***Q*** in the database state that results from **schedule *S*.**

Conflict Serializability

- Thus, only in the **case** where **both *Ii* and *Ij*** are **read instructions** does the relative **order of their execution not matter.**
- We say that ***Ii*** and ***Ij*** **conflict** if they are **operations** by **different transactions** on the **same data item**, and **at least one of these instructions** is a **write operation**.

Conflict Serializability

- Consider **Schedule 3:**
- The **write(A)** instruction of **T1** **conflicts** with the **read(A)** instruction of **T2.**
- However, the **write(A)** instruction of **T2** does **not conflict** with the **read(B)** instruction of **T1,** because the two instructions access different data items.
- Let **Ii** and **Ij** be **consecutive instructions** of a **schedule S**.
- If **Ii** and **Ij** are instructions of **different transactions** and **Ii** and **Ij** **do not conflict**, then **we can swap the order of Ii and Ij** to produce a new **schedule S'.**
- Since the **write(A)** instruction of **T2** in **Schedule 3** does not conflict with the **read(B)** instruction of **T1**, we can **swap** these instructions to generate an equivalent schedule, **schedule 5.**

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)

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

Conflict Serializability

- We **continue** to **swap non-conflicting instructions**:
  - Swap the read(B) instruction of $T1$ with the read(A) instruction of $T2$.
  - Swap the write(B) instruction of $T1$ with the write(A) instruction of $T2$.
  - Swap the write(B) instruction of $T1$ with the read(A) instruction of $T2$.
- The final result of these swaps, **schedule 6.**

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

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

- Thus, we have shown that **schedule 3** is **equivalent** to a **serial schedule.**
- This **equivalence** implies that, regardless of the initial system state, **schedule 3** will produce the same **final state** as will some **serial schedule.**

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**.
- In our previous examples, **Schedule 1** is **not conflict equivalent** to **Schedule 2.**
- However, **Schedule 1** is **conflict equivalent** to **Schedule 3.**

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

Schedule 1

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

Schedule 2

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 3

Conflict Serializability

- The concept of **conflict equivalence** leads to the concept of **conflict serializability.**
- We say that a concurrent ***schedule S is conflict serializable if it is conflict equivalent to a serial schedule.***
- Thus, **Schedule 3** is **conflict serializable**, since it is **conflict equivalent** to the **serial Schedule 1.**

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

Schedule 1

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 3

Conflict Serializability

- Finally, consider **Schedule 7** of Figure:

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

- It consists of only the significant operations (that is, the read and write) of transactions ***T*3** and ***T*4.**
- This schedule is **not conflict serializable**, since it is not equivalent to either the **serial schedule <*T*3,*T*4>** or the **serial schedule <*T*4,*T*3>.**

Important Note

- It is possible to have **two schedules** that produce the **same outcome**, but that are **not conflict equivalent.**
- For **example,** consider **transaction T5**, which transfers **$10** from account B to account A.

Conflict Serializability

- Let **Schedule 8** be as defined in Figure.

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Schedule 8

- We claim that **Schedule 8** is **not conflict equivalent** to the **serial schedule ,** since, in **Schedule 8,** the **write(B)** instruction of **T5 conflicts** with the **read(B)** instruction of **T1.**
- Thus, we cannot move all the instructions of **T1** before those of **T5** by swapping consecutive non-conflicting instructions.
- However, the final values of accounts **A** and **B** after the execution of either **schedule 8** or the **serial schedule** are the **same** —$960 and $2040, respectively.

Testing for Conflict Serializability

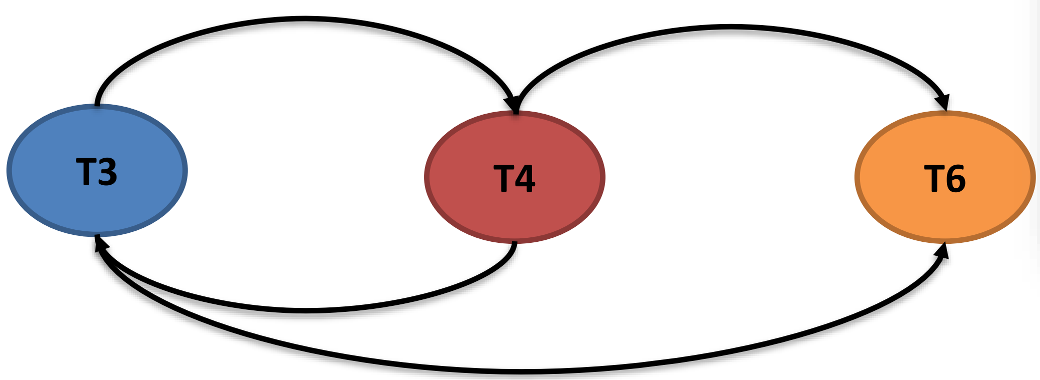
- Create a **precedence graph G(V,E)** where **V** is the set of all the **transactions.**
- We will **create** an **edge Ti → Tj** if **one** of the **three condition is true**:

Ti	Tj
W(Q)	
	.... R(Q)

Ti	Tj
R(Q)	
	.... W(Q)

Ti	Tj
W(Q)	
	.... W(Q)

- The **schedule S** is **serializable** if and only if the **precedence graph G(V,E) has no cycles.**



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

Schedule 9

Testing for Conflict Serializability

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

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);	

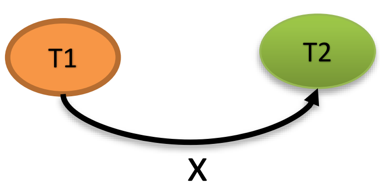
Schedule B

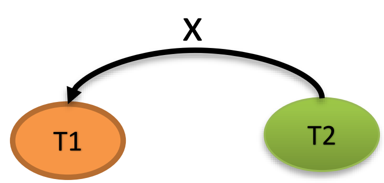
T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

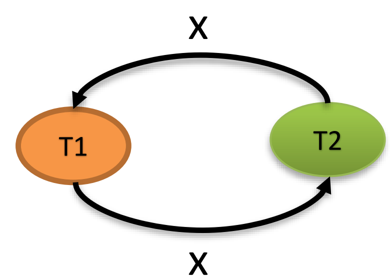
Schedule C

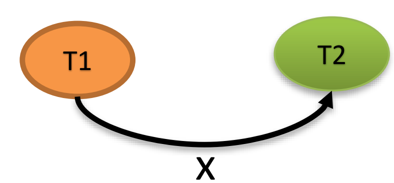
T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

Schedule D









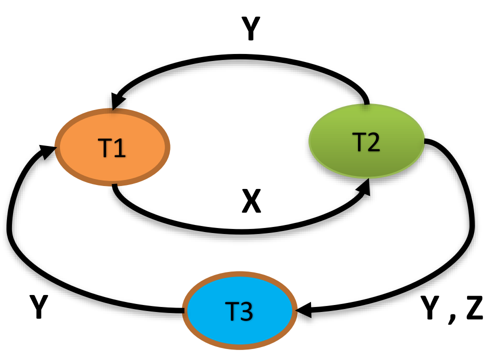
Testing for Conflict Serializability

Transaction T_1	Transaction T_2	Transaction T_3

	read_item(Z);
	read_item(Y);
	write_item(Y);	

		read_item(Y); read_item(Z);
read_item(X); write_item(X);		
		write_item(Y); write_item(Z);
	read_item(X);	
read_item(Y); write_item(Y);		
	write_item(X);	

Schedule E



Equivalent serial schedules

None

Reason

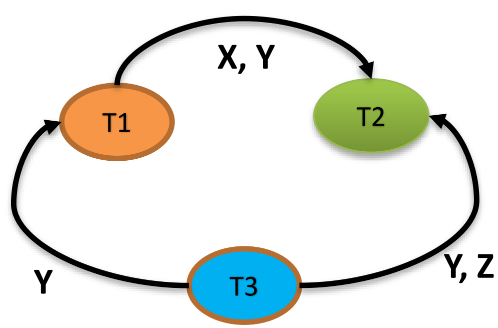
Cycle $X(T_1 \rightarrow T_2), Y(T_2 \rightarrow T_1)$

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

Testing for Conflict Serializability

Transaction T_1	Transaction T_2	Transaction T_3
		read_item(Y);
		read_item(Z);
read_item(X);		
write_item(X);		
		write_item(Y);
		write_item(Z);
	read_item(Z);	
read_item(Y);		
write_item(Y);		
	read_item(Y);	
	write_item(Y);	
	read_item(X);	
	write_item(X);	

Schedule F

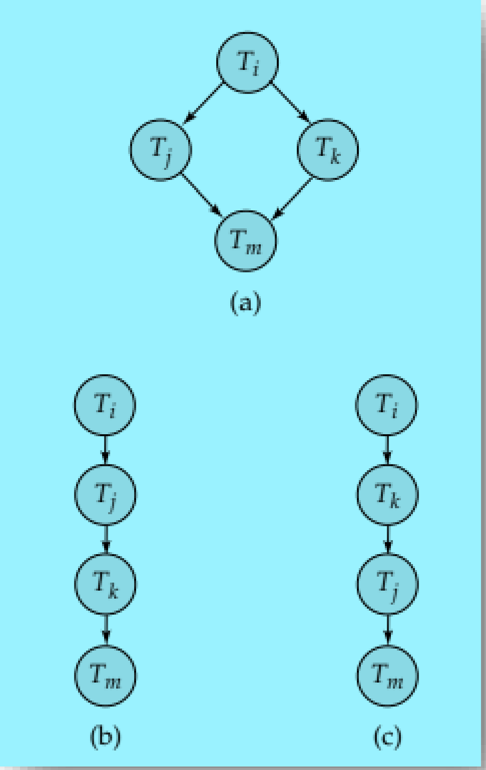


Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Serializability Order

- A **Serializability order** of the **Transactions** can be obtained through **Topological Sorting.**



View Serializability

- **View equivalence:**
- Consider two **schedules S** and **S'** , where the same set of **transactions** participates in both schedules.
- The **schedules S** and **S'** are said to be **view equivalent** if **three conditions** are met:
- **Condition 1:**
- For each **data item Q**, if transaction **Ti reads** the **initial value of Q** in **schedule S,** then **transaction Ti** must, in **schedule S**' , also **read the initial value of Q.**

Schedule S	
Ti	**Tj**
Read(Q)	
	Read(Q)

Schedule S'	
Ti	**Tj**
Read(Q)	
	Read(Q)

View Serializability

- **Condition 2:**
- For each **data item Q**, if transaction **Ti executes read(Q) in schedule S**, and if that value was produced by a **write(Q) operation** executed by **transaction Tj** , then the **read(Q) operation of transaction Ti** must, in **schedule S'** , also read the value of **Q** that was produced by the same **write(Q) operation** of **transaction Tj .**

Schedule S	
Ti	**Tj**
	Write(Q)
Read(Q)	

Schedule S'	
Ti	**Tj**
	Write(Q)
Read(Q)	

View Serializability

- **Condition 3:**
- For each **data item Q**, the transaction (if any) that performs the **final write(Q) operation in schedule S** must perform the **final write(Q)** operation in **schedule S'.**

Schedule S	
Ti	**Tj**
Read(Q) **Write(Q)**	
	Write(Q)
Write(Q)	

Schedule S'	
Ti	**Tj**
	Read(Q) **Write(Q)**
Read(Q) **Write(Q)**	

View Serializability

- **Conditions 1** and **2** ensure that each **transaction reads the same values** in both schedules and, therefore, performs the same computation.
- **Condition 3,** coupled with **conditions 1 and 2**, ensures that **both schedules result** in the **same final system state.**

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

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

- **Schedule 1** is **not view equivalent** to **Schedule 2**, since, in **schedule 1**, the value of **account A** read by **transaction T2** was **produced by T1,** whereas this case does not hold in **Schedule 2.**

View Serializability

- However, **schedule 1** is **view equivalent** to **schedule 3**, because the values of account A and B read by transaction T2 were produced by T1 in both schedules.

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

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)

- The concept of **view equivalence** leads to the concept of **View Serializability.**
- We say that a **schedule S** is **view serializable** if it is **view equivalent** to a **serial schedule.**

View Serializability

- **Schedule 9** is **view serializable** as it is **view equivalent** to the **serial schedule <T3, T4, T6>,** since the one read(Q) instruction reads the initial value of Q in both schedules, and T6 performs the final write of Q in both schedules.

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

Schedule 9

- Every **Conflict-serializable** schedule is **also View serializable**, but there are **View serializable schedules** that are **not Conflict serializable.**
- Indeed, **Schedule 9** is **not conflict serializable**, since every pair of consecutive instructions conflicts, and, thus, **no swapping of instructions is possible**.
- In **Schedule 9**, **transactions T4** and **T6** perform **write(Q) operations** without having performed a **read(Q) operation.**
- Writes of this sort are called **blind writes.**
- **Blind writes** appear in any **view-serializable schedule** that is **not conflict serializable**.

Testing for View Serializability

- **Testing** for **View Serializability** is rather **complicated.**
- In fact, it has been shown that the **problem of testing for view serializability is** itself **NP-complete.**
- Thus, **almost certainly there exists no efficient algorithm** to **test for View Serializability.**

Recoverable Schedules

- Consider **schedule 11** in Figure below, in which **T9** is a transaction that performs only one instruction**: read(A).**

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

- Suppose that the system allows **T9 to commit** immediately after executing the **read(A) instruction.**
- Thus, **T9 commits before T8** does.
- Now suppose that **T8 fails** before it commits.
- Since **T9** has **read** the value of data item **A** written by **T8**, we must **abort T9** to ensure transaction **atomicity.**
- However, **T9** has **already committed** and **cannot be aborted.**

Recoverable Schedules

- Thus, we have a situation where it is **impossible to recover correctly from the failure of T8.**
- **Schedule 11,** with the **commit** happening **immediately after** the **read(A)** instruction, is an **example** of a **Non-recoverable schedule** , which **should not be allowed.**
- A **Recoverable schedule** is one where, for **each pair of transactions Ti and Tj** such **that :**
  - **Tj reads a data item previously written by Ti.**
  - The **Commit operation of Ti** appears **before the Commit operation of Tj .**

T_8	T_9
read(A)	
write(A)	
	read(A)
	write(A)
Commit	
	Commit

Cascadeless Schedules

- Sometimes to **recover** correctly from the failure of a transaction **Ti**, we may have to **roll back several transactions**.
- **Example:** Consider the partial schedule of Figure:
- **Transaction T10** writes a value of **A** that is read by **transaction T11.**
- **Transaction T11** writes a value of **A** that is read by **transaction T12.**
- Suppose that, at this point, **T10 fails.**
- **T10 must be rolled back.**
- Since **T11 is dependent** on **T10, T11 must be rolled back.**
- Since **T12 is dependent on T11**, **T12 must be rolled back.**
- This **phenomenon**, in which a **single transaction failure leads to a series of transaction rollbacks, is called cascading rollback.**

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

Cascadeless Schedules

- **Cascading rollback is undesirable**, since it leads to the **undoing** of a significant amount of work.
- It is **desirable to restrict** the schedules to those where **cascading rollbacks** cannot occur.
- Such schedules are called **Cascadeless Schedules**.
- Formally, a **Cascadeless Schedule** is one where**, for each pair of transactions Ti and Tj such that Tj reads a data item previously written by Ti, the commit operation of Ti appears before the read operation of Tj** .
- **Every Cascadeless** schedule is also **Recoverable.**