Transaction

Chapters 15: Transaction Management

- Transaction (Chapter 15)
  - Transaction Concept
  - Transaction State
  - Concurrent Executions
  - Serializability
  - Recoverability
  - Testing for Serializability

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. transaction to transfer €50 from account A to account B:
  1. **read_from_acoount**(*A*)
  2. *A* := *A* – 50
  3. **write_to_account**(*A*)
  4. **read_from_accont**(*B*)
  5. *B* := *B* + 50
  6. **write_to_account**(*B)*
- Two main issues to deal with:
  - Failures of various kinds, such as **hardware failures and system crashes**
  - Concurrent execution of **multiple transactions**

Transaction ACID properties

- E.g. transaction to transfer €50 from account A to account B:
  1. **read_from_acoount**(*A*)
  2. *A* := *A* – 50
  3. **write_to_account**(*A*)
  4. **read_from_accont**(*B*)
  5. *B* := *B* + 50
  6. **write_to_account**(*B)*
- **Atomicity requirement**
  - if the transaction fails **after step 3 and before step 6**, money will be “lost” leading to an inconsistent database state
    - Failure could be due to software or hardware
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
  - **All or nothing**, regarding the execution of the transaction
- **Durability requirement** — once the user has been notified **of transaction has completio**n, the **updates must persist** in the database even if there are **software or hardware failures**.

Transaction ACID properties (Cont.)

- Transaction to transfer €50 from account A to account B:
  1. **read_from_acoount**(*A*)
  2. *A* := *A* – 50
  3. **write_to_account**(*A*)
  4. **read_from_accont**(*B*)
  5. *B* := *B* + 50
  6. **write_to_account**(*B)*
- **Consistency requirement** in above example:
  - **the sum of A and B** is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g**. sum of balances of all accounts**, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database and must leave a consistent database
  - During transaction execution the database may be **temporarily inconsistent**.
    - Constraints to be v**erified** only at the end of the transaction

Transaction ACID properties (Cont.)

- **Isolation requirement** — if between **steps 3 and 6**, another transaction T2 is allowed to access the **partially updated database**, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	**T2**
1. **read**(*A*)	
2. $A := A - 50$	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. $B := B + 50$	
6. **write**(*B*	

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties - Summary

A **transaction** is a **unit of program execution** that accesses and possibly updates various data items.To preserve the **integrity of data** the database system must ensure:

- **Atomicity** Either **all operations** of the transaction are properly reflected in the database or **none are.**
- **Consistency** Execution of a (single) transaction preserves the **consistency of the database**.
- **Isolation** Although multiple transactions may **execute concurrently**, each transaction must be unaware of other concurrently executing transactions. **Intermediate transaction results** must be hidden from other concurrently executed transactions.
  - That is, 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**.

Non-ACID Transactions

- There are application domains where ACID properties are not necessarily desired or, most likely, not always possible.
- This is the case of so-called **long-duration transactions**
  - Suppose that a transaction takes a lot of time
  - In this case it is **unlikely** that isolation can/should be guaranteed
    - E.g. Consider a transaction of booking a hotel and a flight
- Without Isolation, **Atomicity may be compromised**
- **Consistency and Durability should be preserved**

- Usual solution for long-duration transaction is to define **compensation action** – **what to do if later the transaction fails**
- In (centralized) databases long-duration transactions are usually not considered.
- But these are more and more important, specially in the **context of the Web.**

Transaction State

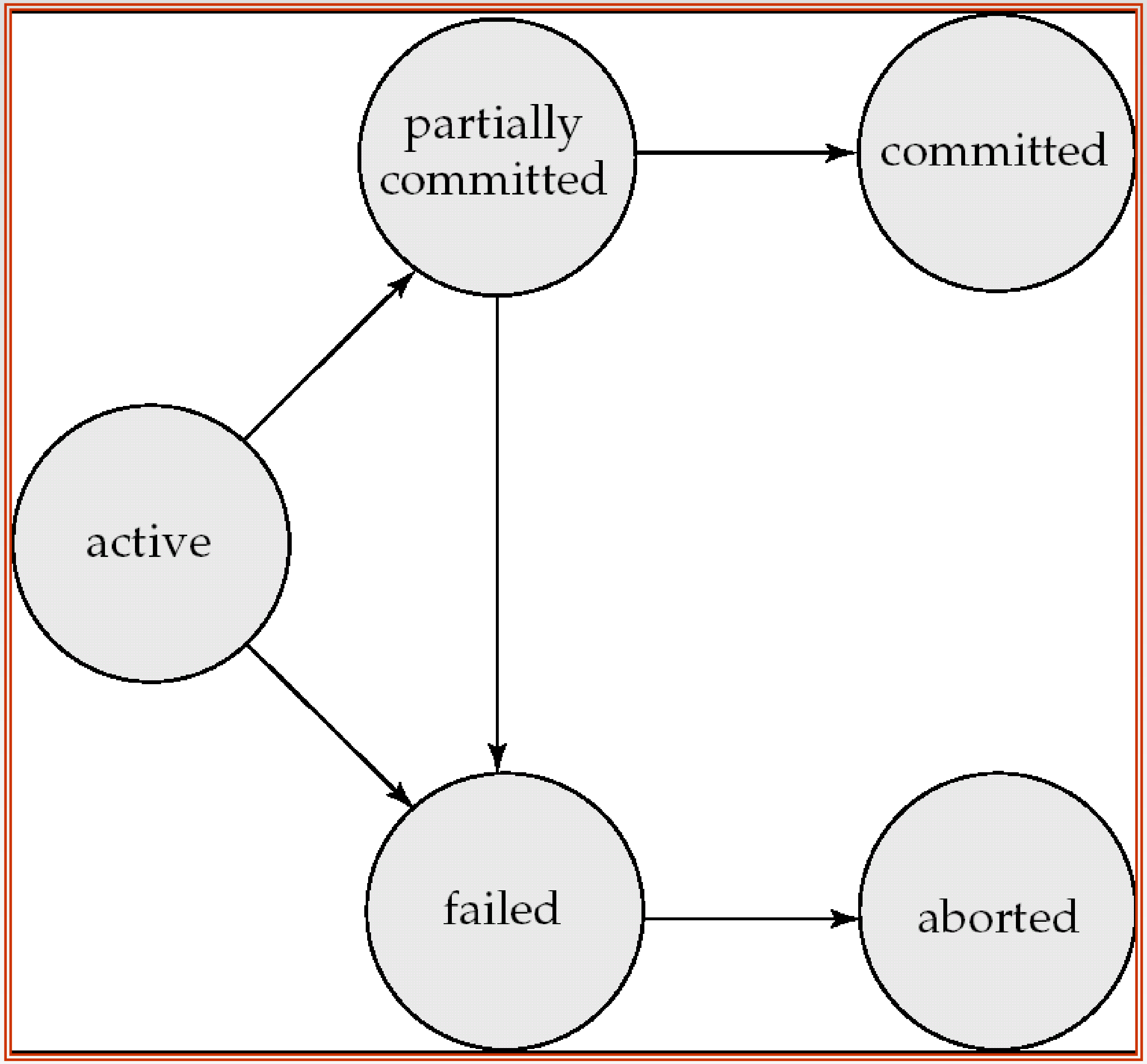
- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the **final statement** has been executed.
- **Failed** – after the discovery that **normal execution** can **no longer proceed.**
- **Aborted** – after the transaction has been **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
    - can be done only if no internal logical error
  - **kill** the transaction
- **Committed** – after **successful completion.**

- To guarantee atomicity, **external observable action** should all be performed (in order) after the **transaction is committed**.

Committed Transactions

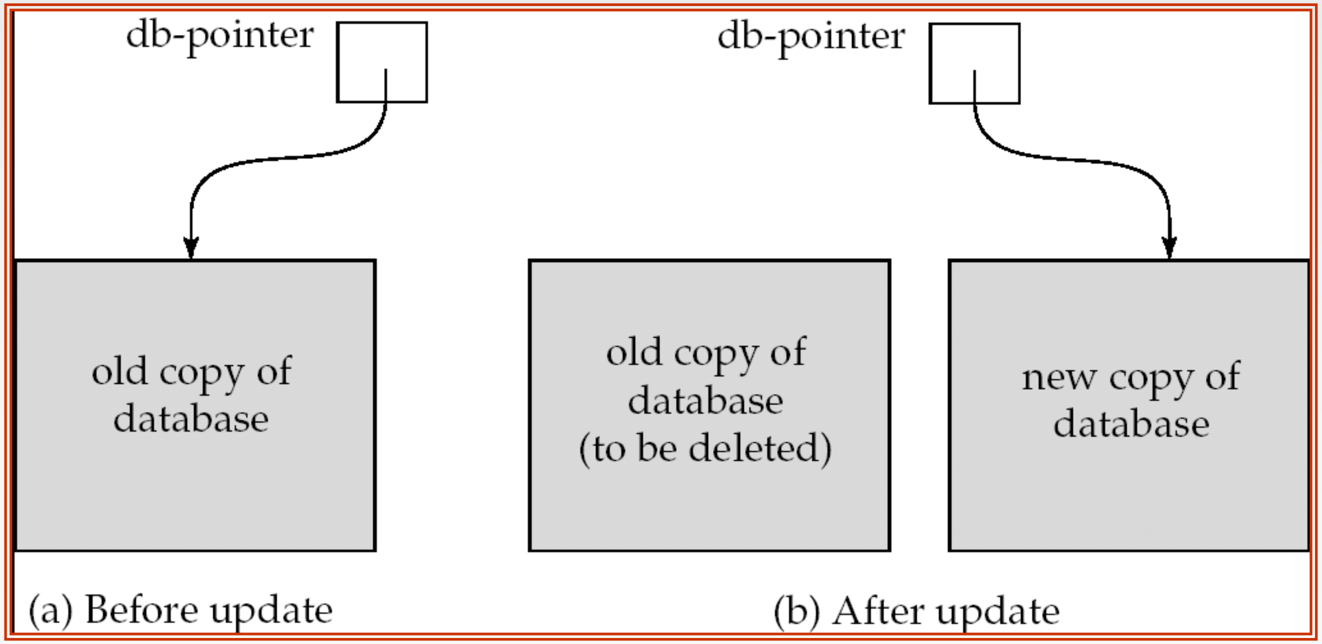
- A committed transaction that has performed updates transforms the database into anew **consistent state**, which **must persist** even if there is a **system failure.**
- It cannot undo its effects by aborting it
- The only way to undo the effects of a committed transaction is to execute a **compensating transaction.**

Transaction State (Cont.)



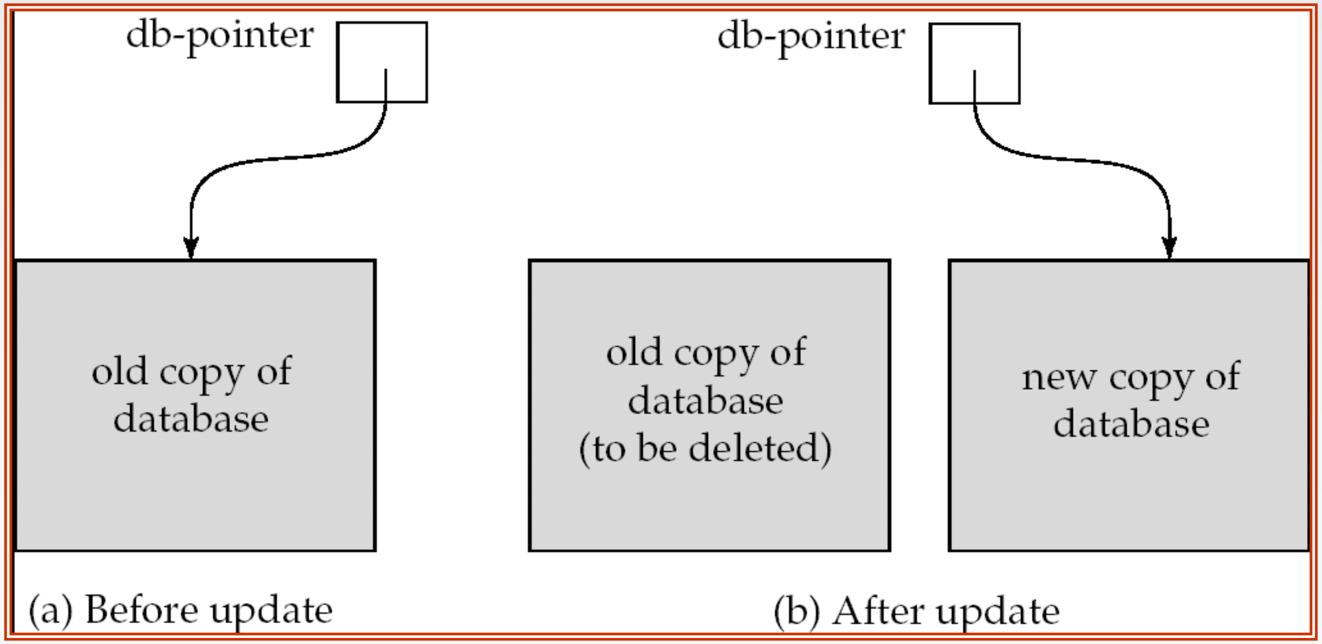
Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the ***shadow-database*** scheme:
  - all updates are made on a ***shadow copy*** of the database
    - **db_pointer** is made to point to the updated shadow copy after
      - the transaction reaches partial commit and
      - all updated pages have been flushed to disk.



Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the ***shadow-database*** scheme:
  - all updates are made on a ***shadow copy*** of the database
    - **db_pointer** is made to point to the updated shadow copy after
      - the transaction reaches partial commit and
      - after the operating system has written all the pages to disk



Implementation of Atomicity and Durability (Cont.)

- db_pointer always points to the current consistent copy of the database.
  - In case transaction fails, **old consistent** copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.
- The shadow-database scheme:
  - Assumes that **only one transaction** is active at a time.
  - Assumes **disks do not fail**
  - Useful for text editors, but
    - extremely **inefficien**t for **large databases**(!)
      - Variant called shadow paging reduces copying of data, but is still not practical for large databases
  - Does not handle c**oncurrent transactions**
- Other implementations of atomicity and durability are possible, e.g. by using logs.
  - Log-based recovery will be addressed later.

Schedules

- **Schedule** – a sequences of instructions that specify the **chronological order** in which instructions of **concurrent transactions are executed**
  - a schedule for a **set of transactions** must consist of all instructions of those transactions
  - **must preserve the order** in which the instructions appear in each individual transaction.
- 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

- The goal is to find schedules that preserve the consistency.

Example Schedule 1

- Let T_1 transfer €50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

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)

Example Schedule 2

- A serial schedule where T_2 is followed by T_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)	

Example Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to 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)

In Schedules 1, 2 and 3, **the sum A + B is preserved**.

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

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

Serializability

- **Goal :** Deal with concurrent schedules that are equivalent to some serial execution:
  - **Basic Assumption** – Each transaction preserves **database consistency.**
  - Thus **serial execution of a set of transactions** preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**
- *Simplified view of transactions*
  - We **ignore operations** other than **read** and **write** instructions
  - We assume that transactions **may perform arbitrary computations** on data in **local buffers** in **between reads and writes.**
  - Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least **one of these instructions wrote *Q***.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces an order between them.
  - **If l_i and l_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.

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

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

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

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

Continue to Swap Nonconflicting 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**.

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

Conflict Equivalent.

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

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
- 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 it 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

Schedule 1 is not conflict equivalent to Schedule 2

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

Schedule 1 is conflict equivalent to Schedule 3

read(B) and write(B) instruction of T1 **>swapped** read(A) and write(A) instruction of T2.

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 (Cont.)

- Example of a **schedule-7** 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 < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Schedules that produce the same outcome But that are not conflict equivalent.

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)

Figure 15.11 Schedule 8.

View Serializability

- Sometimes it is **possible to serialize schedules that are not conflict** serializable
- View serializability provides a weaker and still consistency preserving notion of serialization
- 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 schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. 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'.

Schedule 1 is not VIEW equivalent to Schedule 2

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

Schedule 1 is conflict equivalent to Schedule 3

read(B) and write(B) instruction of T1 **>swapped** read(A) and write(A) instruction of T2.

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

View Serializability (Cont.)

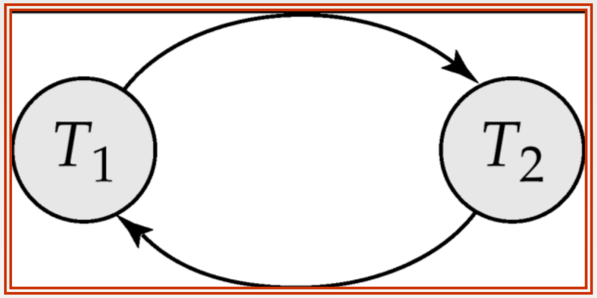
- A schedule *S* is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.

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

- It is equivalent to either <T3,T4,T6> or <T4,T3,T6>
- Every view serializable schedule that is not conflict serializable has **blind writes.**

Testing for Serializability (Precedence Graph)

1. T_i executes write(Q) before T_j executes read(Q).
2. T_i executes read(Q) before T_j executes write(Q).
3. T_i executes write(Q) before T_j executes write(Q).



Example Schedule (Schedule A) + 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)