Transaction Processing

Dr. Jyotismita Chaki

Introduction to Transaction Processing

- A transaction is an executing program that forms a logical unit of database processing.
- A transaction includes one or more database access operations—these can include insertion, deletion, modification (update), or retrieval operations.
- Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions.
- Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications.
- These systems require high availability and fast response time for hundreds of concurrent users.

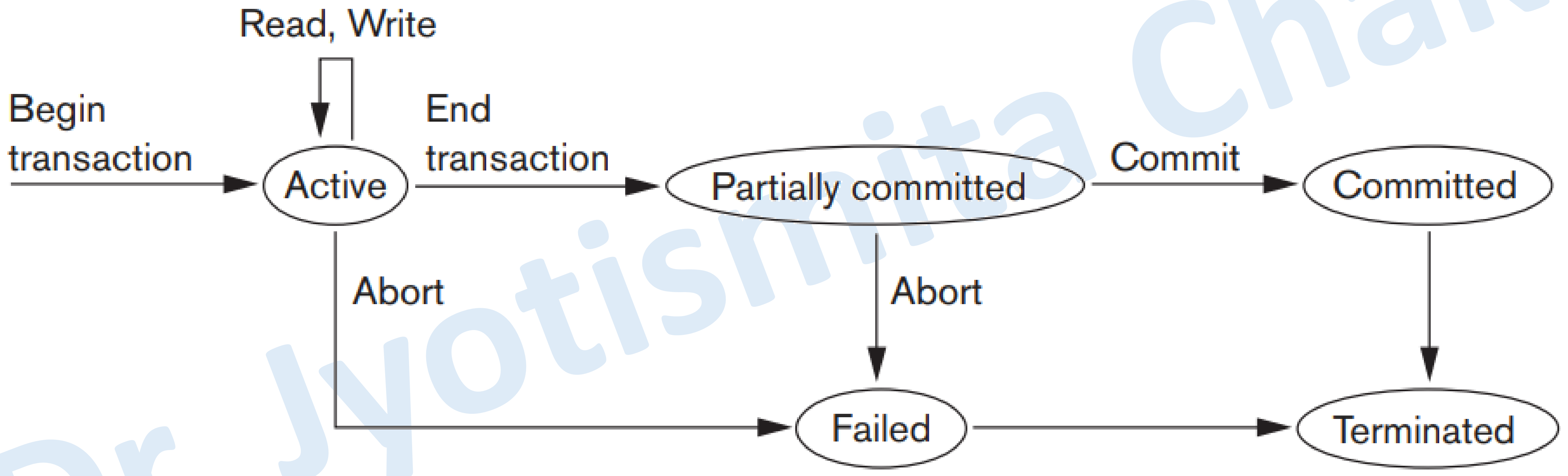
Transaction and System Concepts

- A transaction is an atomic unit of work that should either be completed in its entirety or not done at all.
- For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts.
- Therefore, the recovery manager of the DBMS needs to keep track of the following operations:
  - BEGIN_TRANSACTION. This marks the beginning of transaction execution.
  - READ or WRITE. These specify read or write operations on the database items that are executed as part of a transaction.
  - END_TRANSACTION. This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution.
  - COMMIT_TRANSACTION. This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
  - ROLLBACK (or ABORT). This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Transaction and System Concepts

- A transaction goes into an active state immediately after it starts execution, where it can execute its **READ and WRITE operations**.
- When the transaction ends, it moves to the **partially committed state**.
- At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not.
- If these checks are successful, the transaction is said to have reached its commit point and enters the **committed state**.
- However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state.
- The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database.
- The terminated state corresponds to the transaction leaving the system.

Transaction and System Concepts



State transition diagram illustrating the states for transaction execution

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, as well as other transaction information that may be needed to permit recovery from failures.
- 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, called the **log buffers**, hold the last part of the log file, so that log entries are first added to the **log 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 System Log

- 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.
- In these entries, T refers to a unique transaction-id that is generated automatically by the system for each transaction and that is used to identify each transaction:
  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.

Desirable Properties of Transactions

- Transactions should possess several properties, often called the ACID properties; they should be enforced by the concurrency control and recovery methods of the DBMS.
- The following are the ACID properties:
  - Atomicity. A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.
  - Consistency preservation. A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.
  - Isolation. A transaction should appear as though it is being executed in isolation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
  - Durability or permanency. The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Desirable Properties of Transactions

- The **atomicity** property requires that we execute a transaction to completion.
- It is the responsibility of the transaction recovery subsystem of a DBMS to ensure atomicity.
- If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database.
- On the other hand, write operations of a committed transaction must be eventually written to disk.

Desirable Properties of Transactions

- The **preservation of consistency** is generally considered to be the responsibility of the programmers who write the database programs and of the DBMS module that enforces integrity constraints.
- A consistent state of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold.
- A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the complete execution of the transaction, assuming that no interference with other transactions occurs.

Desirable Properties of Transactions

- The **isolation** property is enforced by the concurrency control subsystem of the DBMS.
- If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems.
- The **durability** property is the responsibility of the recovery subsystem of the DBMS.

Characterizing schedules based on recoverability

- When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or **history**).
- Operations from different transactions can be interleaved in the schedule S.
- However, for each transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i.
- The order of operations in S is considered to be a total ordering, meaning that for any two operations in the schedule, one must occur before the other.
- It is possible theoretically to deal with schedules whose operations form partial orders, but we will assume for now total ordering of the operations in a schedule.

Characterizing schedules based on recoverability

- For the purpose of recovery and concurrency control, we are mainly interested in the **read_item (**Reads a database item named X into a program variable also named X**)** and **write_item (**Writes the value of program variable X into the database item named X**)** operations of the transactions, as well as the **commit** and **abort** operations.
- A shorthand notation for describing a schedule uses the symbols b, r, w, e, c, and a for the operations **begin_transaction**, **read_item**, **write_item**, **end_transaction**, **commit**, and **abort**, respectively, and appends as a subscript the transaction id (transaction number) to each operation in the schedule.
- In this notation, the database item X that is read or written follows the **r** and **w** operations in parentheses.
- In some schedules, we will only show the read and write operations, whereas in other schedules we will show additional operations, such as commit or abort.

Characterizing schedules based on recoverability: Interpretation of shorthand notations

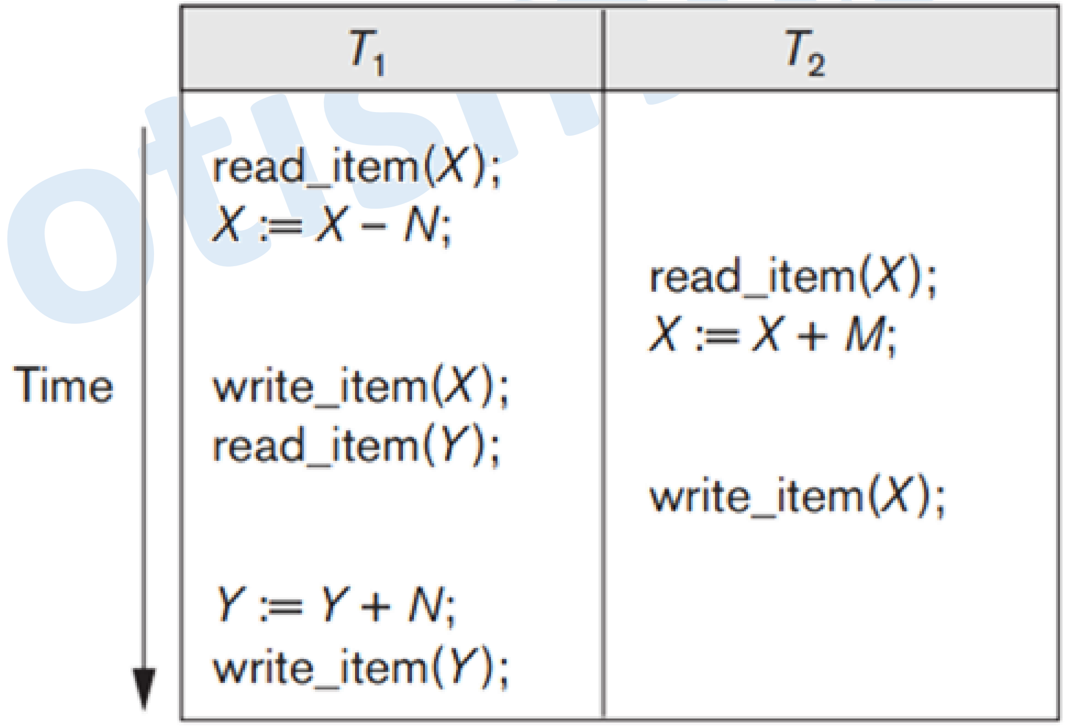
- $b_{\text{<Transaction id>}}$ → E.g: b_1 [begin Transaction T_1].
- $r_{\text{<Transaction id>}}$(<data item>) → E.g: $r_1(X)$ [Transaction T_1 is performing read_item on data item X].
- $w_{\text{<Transaction id>}}$(<data item>) → E.g: $w_1(X)$ [Transaction T_1 is performing write_item on data item X].
- $c_{\text{<Transaction id>}}$ → E.g: c_1 [Transaction T_1 has committed].
- $a_{\text{<Transaction id>}}$ → E.g: a_1 [Transaction T_1 has aborted or rolled back].

Characterizing schedules based on recoverability: read_item and write_item

- Executing a read_item(X) command includes the following steps:
  1. Find the address of the disk block that contains item X.
  2. Copy the disk block into a buffer in main memory if that disk is not already in some main memory buffer.
  3. Copy item X from the buffer to the program variable named X.
- Executing a write_item(X) command includes the following steps:
  1. Find the address of the disk block that contains item X.
  2. Copy the disk block into a buffer in main memory if that disk is not already in some main memory buffer.
  3. Copy item X from the program variable named X into its correct location in the buffer.
  4. Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Characterizing schedules based on recoverability

- The schedule in the following Figure , which we shall call S_a, can be written as follows in this notation:
  - S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;



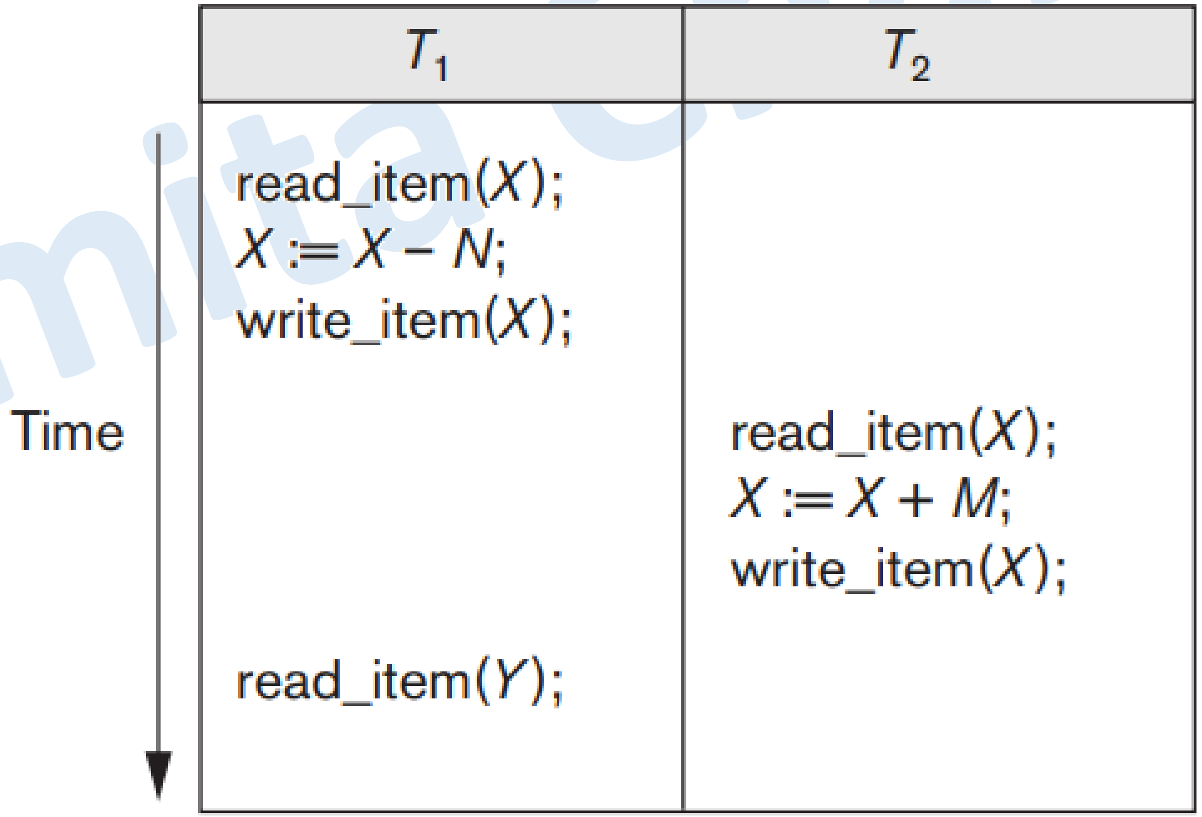
T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

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

Characterizing schedules based on recoverability

- Similarly, the schedule for the following Figure, which we call S_b, can be written as follows, if we assume that transaction T1 aborted after its read_item(Y) operation:
  - S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;



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);	

Characterizing schedules based on recoverability: Conflicting Operations in a Schedule

- Two operations in a schedule are said to **conflict** if they satisfy all three of the following conditions:
  1. they belong to different transactions;
  2. they access the same item X; and
  3. at least one of the operations is a write_item(X).
- For example, in schedule S_a, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$.
- However, the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations; the operations $w_2(X)$ and $w_1(Y)$ do not conflict because they operate on distinct data items X and Y; and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

Characterizing schedules based on recoverability: Conflicting Operations in a Schedule

- Two operations are conflicting if changing their order can result in a different outcome.
- For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, then the value of X that is read by transaction T1 changes, because in the second ordering the value of X is read by $r_1(X)$ after it is changed by $w_2(X)$, whereas in the first ordering the value is read before it is changed. This is called a **read-write** conflict.
- The other type is called a **write-write** conflict where we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$. For a write-write conflict, the last value of X will differ because in one case it is written by T2 and in the other case by T1.
- Two read operations are not conflicting because changing their order makes no difference in outcome.

Characterizing schedules based on recoverability

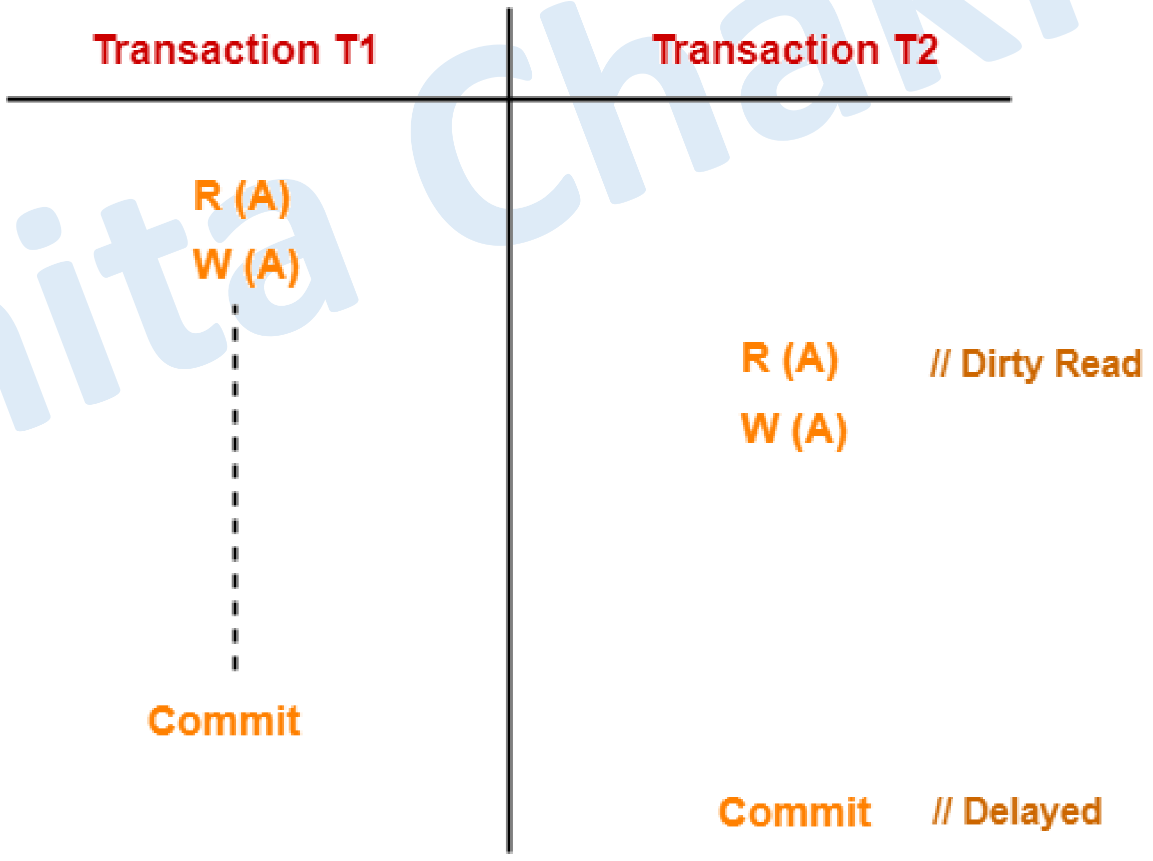
- For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved.
- In some cases, it is even not possible to recover correctly after a failure.
- Hence, it is important to characterize the types of schedules for which *recovery is possible*, as well as those for which *recovery is relatively simple*.
- These characterizations do not actually provide the recovery algorithm; they only attempt to theoretically characterize the different types of schedules.

Characterizing schedules based on recoverability

- Once a transaction T is committed, it should never be necessary to roll back T. This ensures that the durability property of transactions is not violated.
- The schedules that theoretically meet this criterion are called **recoverable schedules**.
- A schedule where a committed transaction may have to be rolled back during recovery is called **nonrecoverable** and hence should not be permitted by the DBMS.

Characterizing schedules based on recoverability

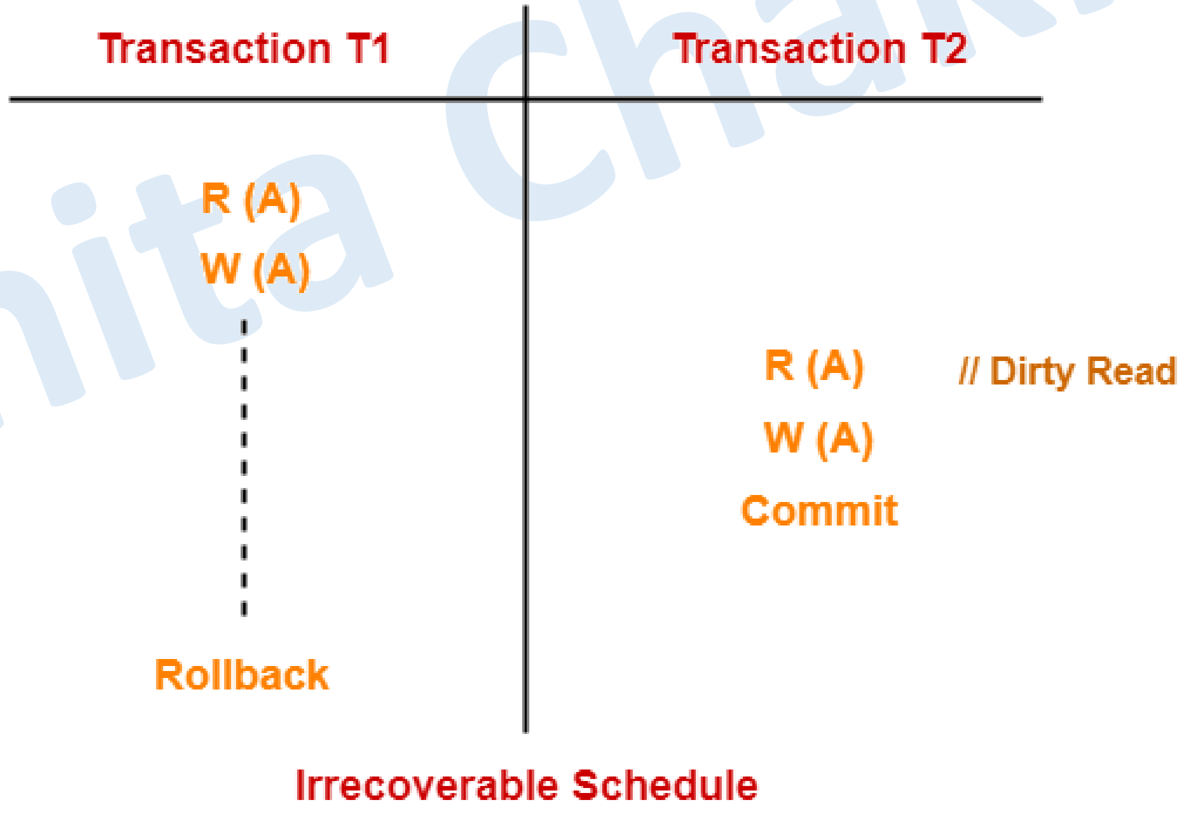
- Consider the following schedule-
- Here,
    - T2 performs a **dirty read** (Reading from an uncommitted transaction is called as a dirty read) operation.
    - The commit operation of T2 is delayed till T1 commits or roll backs.
    - T1 commits later.
    - T2 is now allowed to commit.
    - In case, T1 would have failed, T2 has a chance to recover by rolling back.



Recoverable Schedule

Characterizing schedules based on recoverability

- Consider the following schedule-
- Here
  - T2 performs a **dirty read** operation.
  - T2 commits before T1.
  - T1 fails later and roll backs.
  - The value that T2 read now stands to be incorrect.
  - T2 can not recover since it has already committed.



Characterizing schedules based on recoverability

- The condition for a recoverable schedule is as follows:
    - A schedule S is recoverable if no transaction T in S commits until all transactions T′ that have written some item X that T reads have committed.
    - A transaction T reads from transaction T′ in a schedule S if some item X is first written by T′ and later read by T.
    - In addition, T′ should not have been aborted before T reads item X, and there should be no transactions that write X after T′ writes it and before T reads it (unless those transactions, if any, have aborted before T reads X).
- Some recoverable schedules may require a complex recovery process, but if sufficient information is kept (in the log), a recovery algorithm can be devised for any recoverable schedule.
- The schedules S_a and S_b from the preceding section are both recoverable, since they satisfy the above definition.

Characterizing schedules based on recoverability

- Consider the schedule S_a' given below, which is the same as schedule S_a except that two commit operations have been added to S_a:
  - S_a': $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1;
- S_a' is recoverable, even though it suffers from the lost update problem.
- However, consider the two schedules S_c and S_d that follow:
  - S_c: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1;
- S_c is not recoverable because T2 reads item X from T1, but T2 commits before T1 commits. The problem occurs if T1 aborts after the c_2 operation in S_c; then the value of X that T2 read is no longer valid and T2 must be aborted after it is committed, leading to a schedule that is not recoverable.

Characterizing schedules based on recoverability

- For the schedule to be recoverable, the c_2 operation in S_c must be postponed until after T1 commits, as shown in S_d.
  - S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2;
- If T1 aborts instead of committing, then T2 should also abort as shown in S_e, because the value of X it read is no longer valid.
  - S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2;
- In S_e, aborting T2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule S_c.

Characterizing schedules based on recoverability: Types of recoverable schedule

- In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of a committed transaction as durable is not violated.
- However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, where an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.
- This is illustrated in schedule S_e, where transaction T2 has to be rolled back because it read item X from T1, and T1 then aborted.

Characterizing schedules based on recoverability

- Here,
  - Transaction T2 depends on transaction T1.
  - Transaction T3 depends on transaction T2.
  - Transaction T4 depends on transaction T3.
- In this schedule,
  - The failure of transaction T1 causes the transaction T2 to rollback.
  - The rollback of transaction T2 causes the transaction T3 to rollback.
  - The rollback of transaction T3 causes the transaction T4 to rollback.
  - Such a rollback is called as a **Cascading Rollback**.

T1	T2	T3	T4
R (A)			
W (A)			
	R (A)		
	W (A)		
		R (A)	
		W (A)	
			R (A)
			W (A)
Failure			

Cascading Recoverable Schedule

Characterizing schedules based on recoverability: Types of recoverable schedule

- Because cascading rollback can be time-consuming—since numerous transactions can be rolled back —it is important to characterize the schedules where this phenomenon is guaranteed not to occur.
- A schedule is said to be **cascadeless,** or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions.
- In this case, all items read will not be discarded because the transactions that wrote them have committed, so no cascading rollback will occur.
- To satisfy this criterion, the $r_2(X)$ command in schedules S_d and S_e must be postponed until after T1 has committed (or aborted), thus delaying T2 but ensuring no cascading rollback if T1 aborts.

Characterizing schedules based on recoverability

- Cascadeless schedule allows only committed read operations.
- However, it allows uncommitted write operations.

T1	T2
R (A)	
W (A)	
	W (A) // Uncommitted Write
Commit	

Cascadeless Schedule

T1	T2	T3
R (A)		
W (A)		
Commit		
	R (A)	
	W (A)	
	Commit	
		R (A)
		W (A)
		Commit

Cascadeless Schedule

Characterizing schedules based on recoverability: Types of recoverable schedule

- Finally, there is a third, more restrictive type of schedule, called a **strict schedule**, in which transactions can neither read nor write an item X until the last transaction that wrote X has committed (or aborted).
- Strict schedules simplify the recovery process.
- In a strict schedule, the process of undoing a write_item(X) operation of an aborted transaction is simply to restore the **before image** (old_value or BFIM) of data item X.
- This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules.

Characterizing schedules based on recoverability

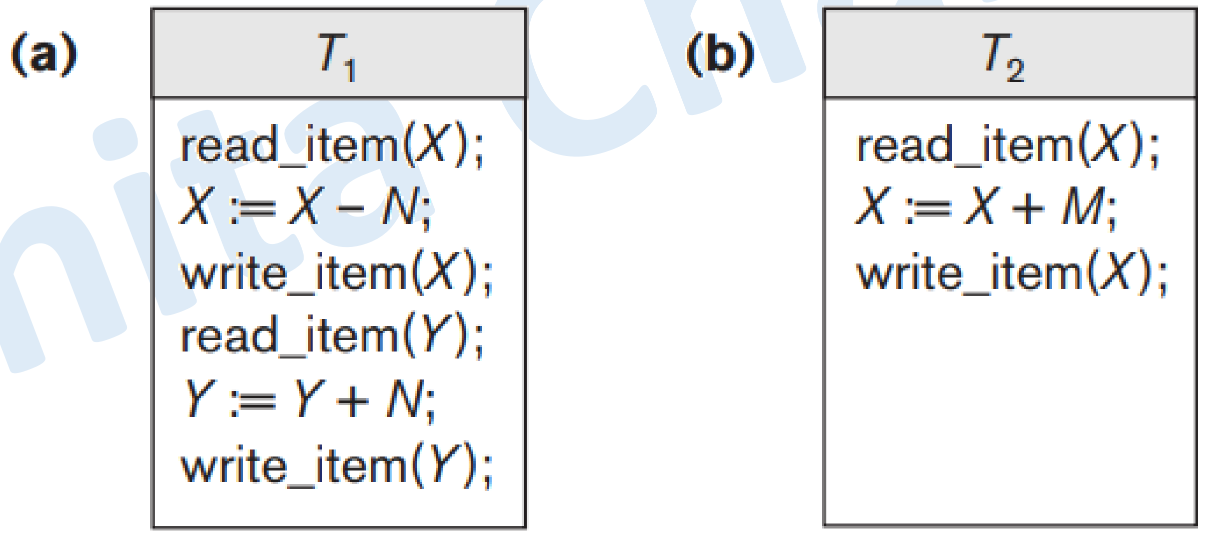
- For example, consider schedule S_f:
  - S_f: $w_1(X, 5)$; $w_2(X, 8)$; a1;
- Suppose that the value of X was originally 5, which is the before image stored in the system log along with the w1(X, 5) operation.
- If T1 aborts, as in S_f, the recovery procedure that restores the before image of an aborted write operation will restore the value of X to 5, even though it has already been changed to 8 by transaction T2, thus leading to potentially incorrect results.
- Although schedule S_f is cascadeless, it is not a strict schedule, since it permits T2 to write item X even though the transaction T1 that last wrote X had not yet committed (or aborted).
- A strict schedule does not have this problem.

Characterizing schedules based on recoverability

- Any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable.
- Suppose we have i transactions T1, T2, … , Ti, and their number of operations are n1, n2, … , ni, respectively.
- If we make a set of all possible schedules of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable.
- The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascadeless schedules.
- Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

Characterizing Schedules Based on Serializability

- A **schedule** (or **history**) S of n transactions T1, T2, … , Tn is an ordering of the operations of the transactions.
- Operations from different transactions can be interleaved in the schedule S.
- We characterize the types of schedules that are always considered to be correct when concurrent transactions are executing.
- Such schedules are known as serializable schedules. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T1 and T2 as shown in Figure at approximately the same time.



(a)

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

(b)

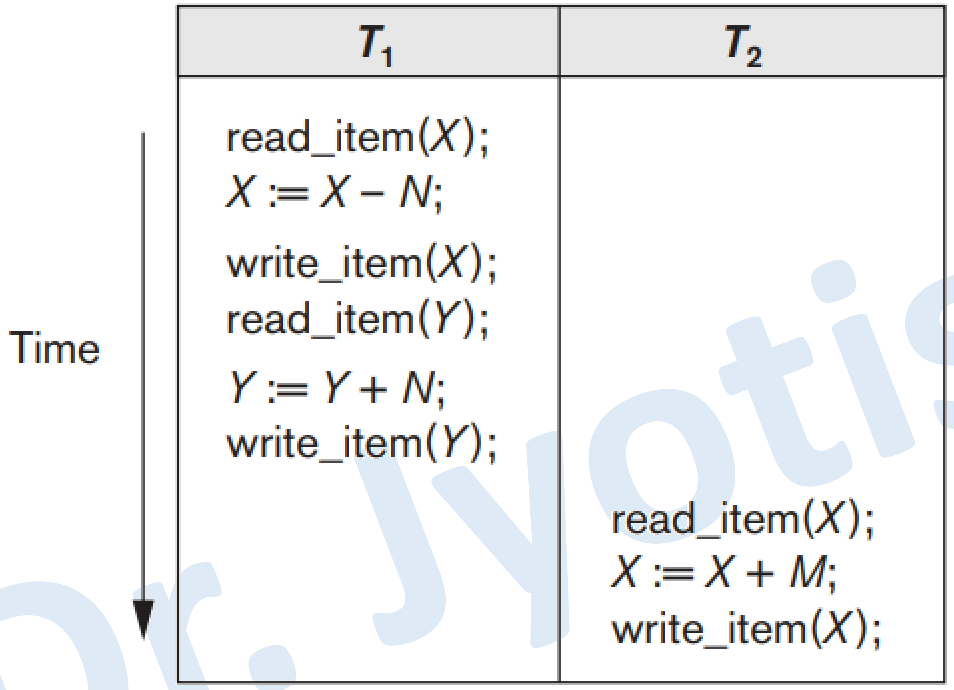
T_2
read_item(X);
$X := X + M$;
write_item(X);

(a) Transaction T1. (b) Transaction T2.

Characterizing Schedules Based on Serializability

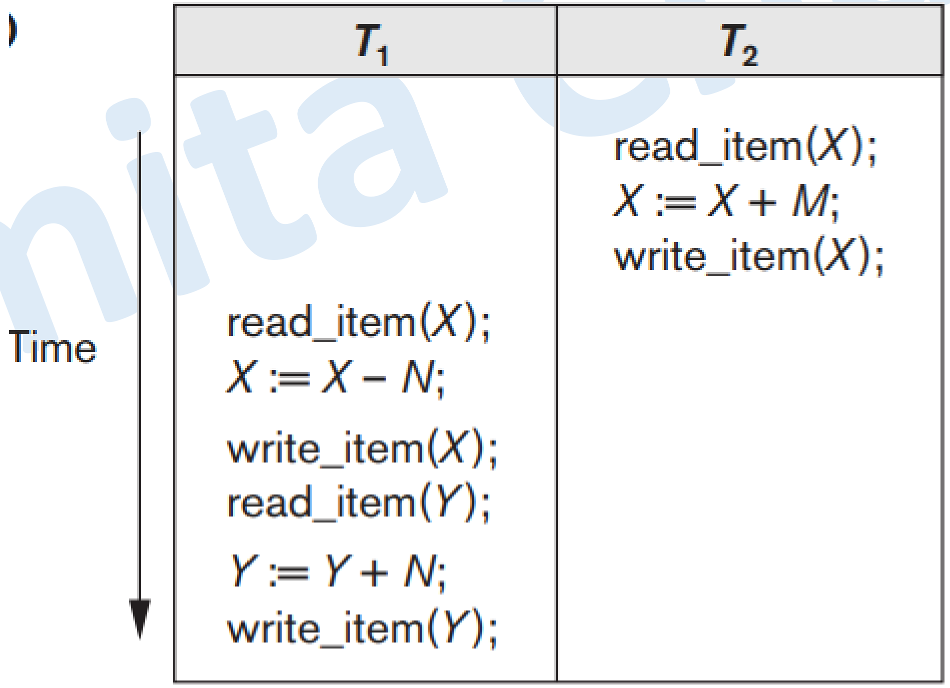
- If no interleaving of operations is permitted, there are only two possible outcomes:
  - Execute all the operations of transaction T1 (in sequence) followed by all the operations of transaction T2 (in sequence).
  - Execute all the operations of transaction T2 (in sequence) followed by all the operations of transaction T1 (in sequence).
- These two schedules—called **serial schedules.**
- Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction.

Characterizing Schedules Based on 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);	

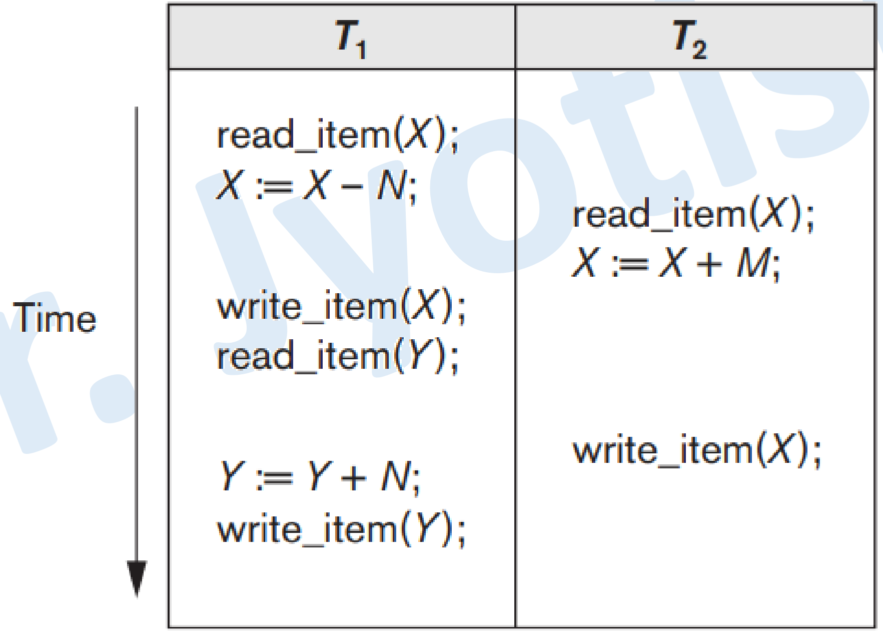
Schedule B

Serial schedule A: T1 followed by T2.

Serial schedule B: T2 followed by T1

Characterizing Schedules Based on Serializability

- If interleaving of operations is allowed, there will be many possible orders (**non-serial schedules**) in which the system can execute the individual operations of the transactions. Two possible schedules are shown in the Figure.

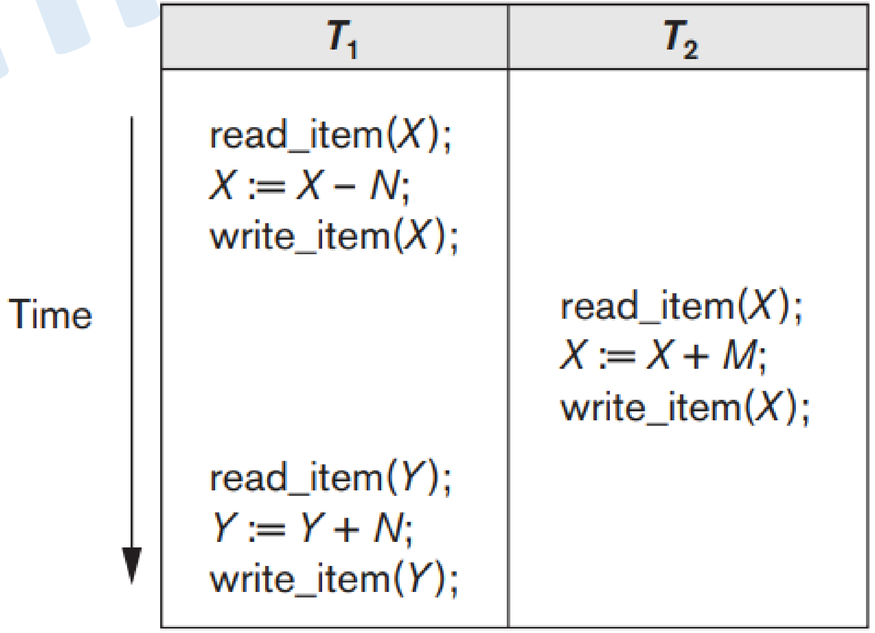


T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

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);	

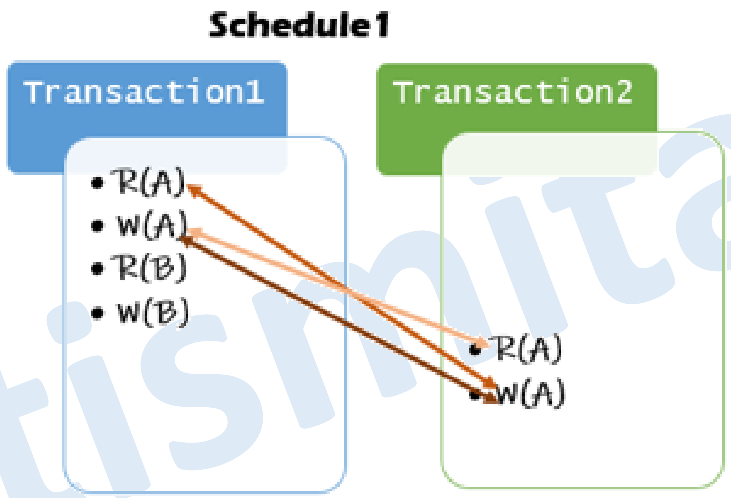
Schedule D

Characterizing Schedules Based on Serializability

- The definition of **serializable schedule** is as follows: A schedule S of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.
- Two schedules are called **result equivalent** if they produce the same final state of the database.
- Two operations in a schedule are said to **conflict** if they belong to different transactions, access the same database item, and either both are write_item operations or one is a write_item and the other a read_item.
- If two **conflicting operations** are applied in different orders in two schedules, the effect can be different on the database or on the transactions in the schedule, and hence the schedules are not **conflict equivalent**.

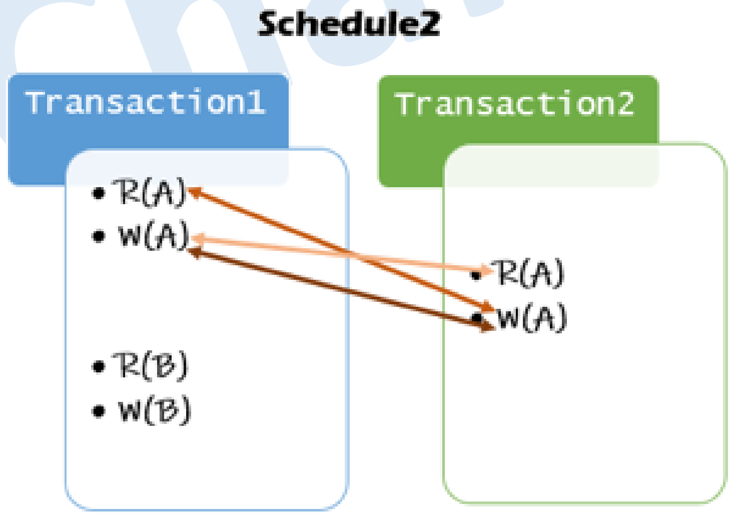
Characterizing Schedules Based on Serializability

- Consider 2 schedules, Schedule1 and Schedule2.
- Schedule2 (a non-serial schedule) is considered to be conflict serializable when its conflict operations are the same as that of Shedule1 (a serial schedule).



Conflicts occurring on data item A during a serial schedule:

1. Read – Write
2. Write – Read
3. Write – Write

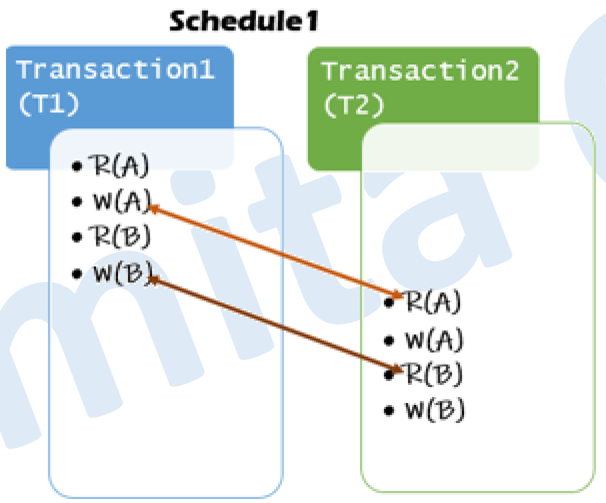


Conflicts occurring on data item A during a non-serial schedule:

1. Read – Write
2. Write – Read
3. Write – Write

Characterizing Schedules Based on Serializability

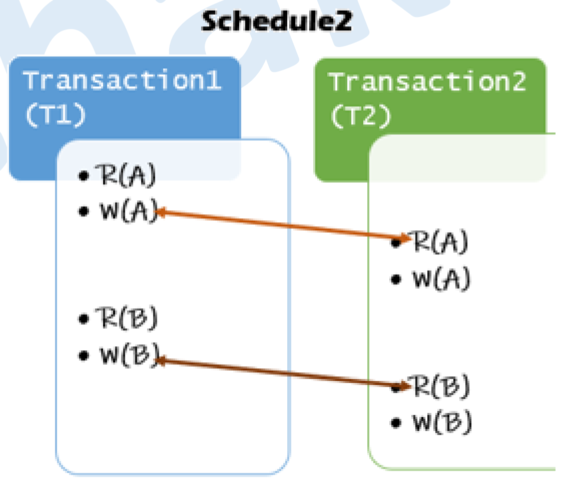
- Two schedules (one being serial schedule and another being non-serial) are said to be **view serializable** if they satisfy the rules for being view equivalent to one another.
- The rules to be upheld are:
  1. Initial values of the data items involved within a schedule must be the same.
  2. Final values of the data items involved within a schedule must be the same.
  3. The number of WR operations performed must be equivalent for the schedules involved.



Initial operation of data item A: **T1**
Initial operation of data item B: **T1**

Final operation of data item A: **T2**
Final operation of data item B: **T2**

Number of WR operations: **2**



Initial operation of data item A: **T1**
Initial operation of data item B: **T1**

Final operation of data item A: **T2**
Final operation of data item B: **T2**

Number of WR operations: **2**

Testing for Serializability of a Schedule

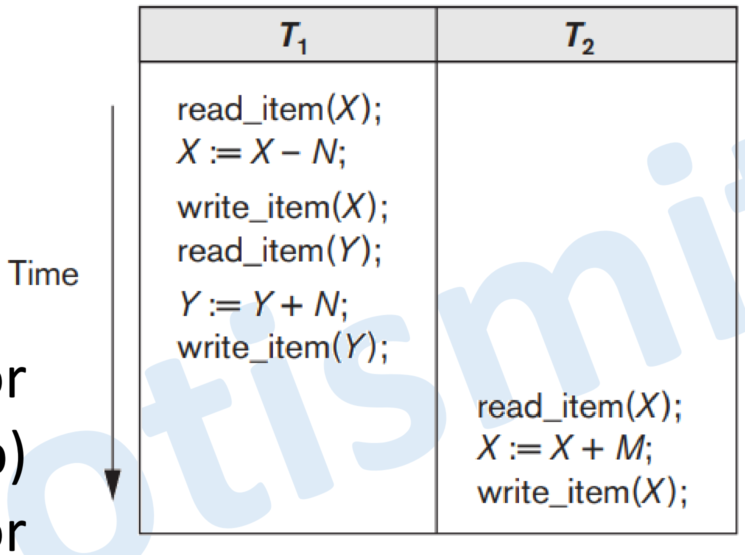
- There is a simple algorithm for determining whether a particular schedule is (conflict) serializable or not.
- The algorithm looks at only the read_item and write_item operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a directed graph $G = (N, E)$ that consists of a set of nodes $N = \{T_1, T_2, \dots , T_n\}$ and a set of directed edges $E = \{e_1, e_2, \dots , e_m\}$.
- There is one node in the graph for each transaction T_i in the schedule. Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, $1 \le j \le n$, $1 \le k \le n$, where T_j is the **starting node** of e_i and T_k is the **ending node** of e_i.
- In the precedence graph, an edge from Ti to Tj means that transaction Ti must come before transaction Tj in any serial schedule that is equivalent to S

Testing for Serializability of a Schedule

- Testing Conflict Serializability of a Schedule S:
  - For each transaction Ti participating in schedule S, create a node labeled Ti in the precedence graph.
  - For each case in S where Tj executes a read_item(X) after Ti executes a write_item(X), create an edge (Ti → Tj) in the precedence graph.
  - For each case in S where Tj executes a write_item(X) after Ti executes a read_item(X), create an edge (Ti → Tj) in the precedence graph.
  - For each case in S where Tj executes a write_item(X) after Ti executes a write_item(X), create an edge (Ti → Tj) in the precedence graph.
  - The schedule S is serializable if and only if the precedence graph has no cycles.

Testing for Serializability of a Schedule

(a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B.

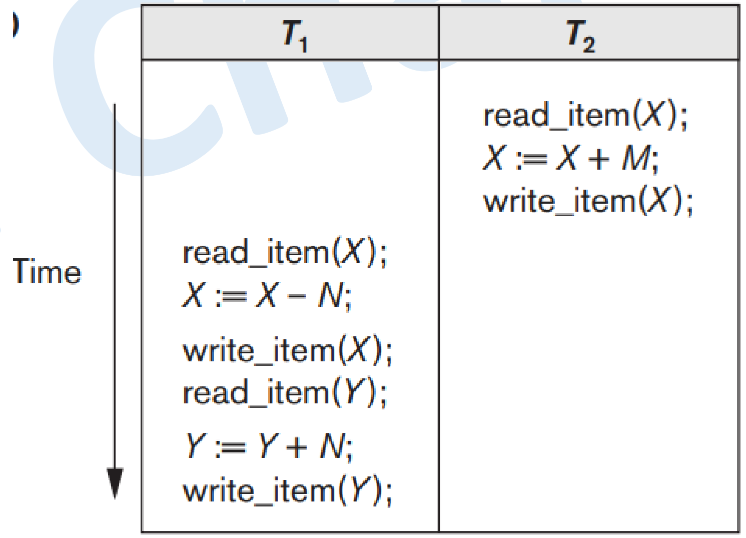


T_1	T_2

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

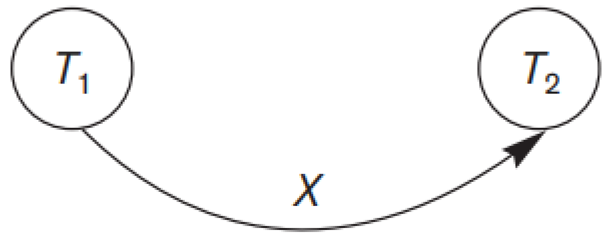


T_1	T_2

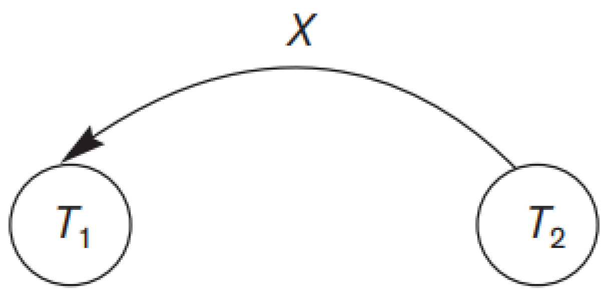
	read_item(X);
	$X := X + M$;
	write_item(X);

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

Schedule B

(a)


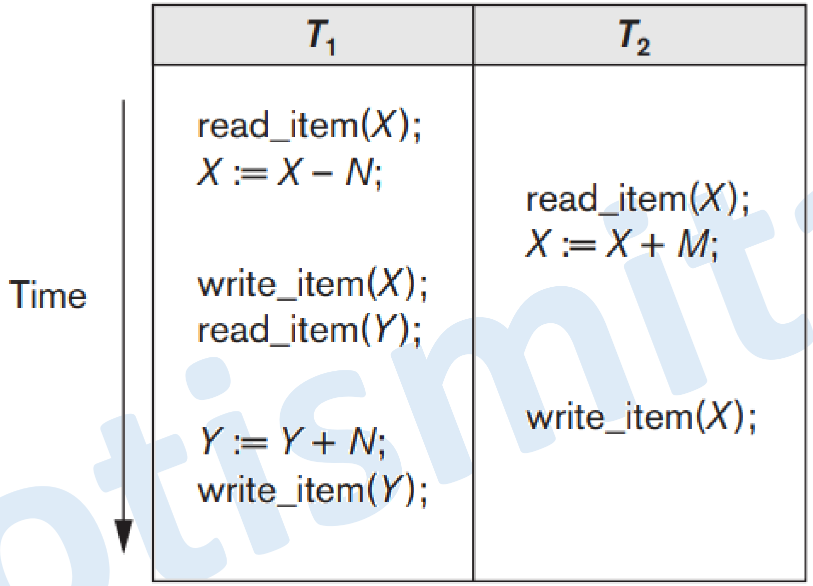
$$\begin{tikzcd} T_1 \arrow[r, bend right, "X"'] & T_2 \end{tikzcd}$$

(b)


$$\begin{tikzcd} T_1 & T_2 \arrow[l, bend right, "X"'] \end{tikzcd}$$

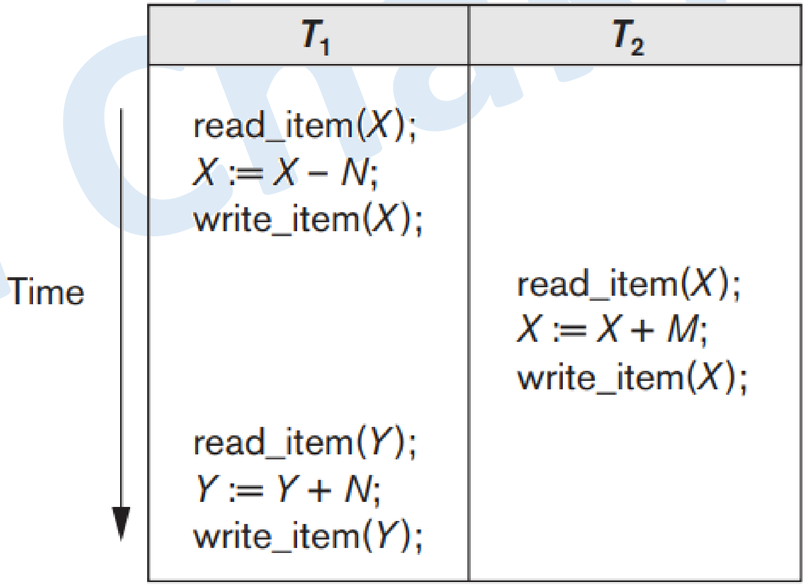
Testing for Serializability of a Schedule

(c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).



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);	

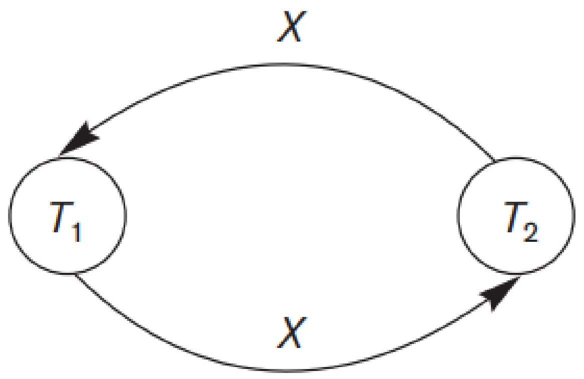
Schedule C



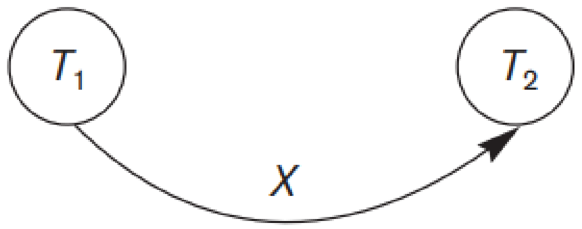
T_1	T_2
read_item(X); $X := X - N$; write_item(X);	
	read_item(X); $X := X + M$; write_item(X);
read_item(Y); $Y := Y + N$; write_item(Y);	

Schedule D

(c)



(d)

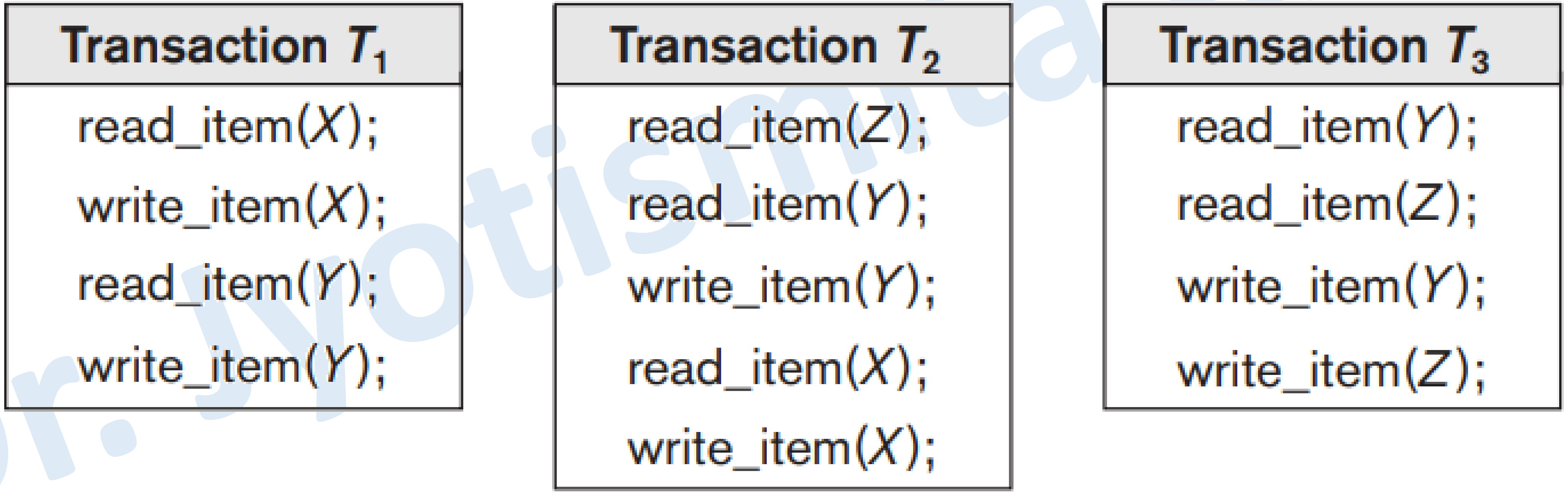


Testing for Serializability of a Schedule

- If the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable.
- The graph for schedule C has a cycle, so it is not serializable.
- The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is T1 followed by T2.
- The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.

Testing for Serializability of a Schedule

- Another example, in which three transactions participate



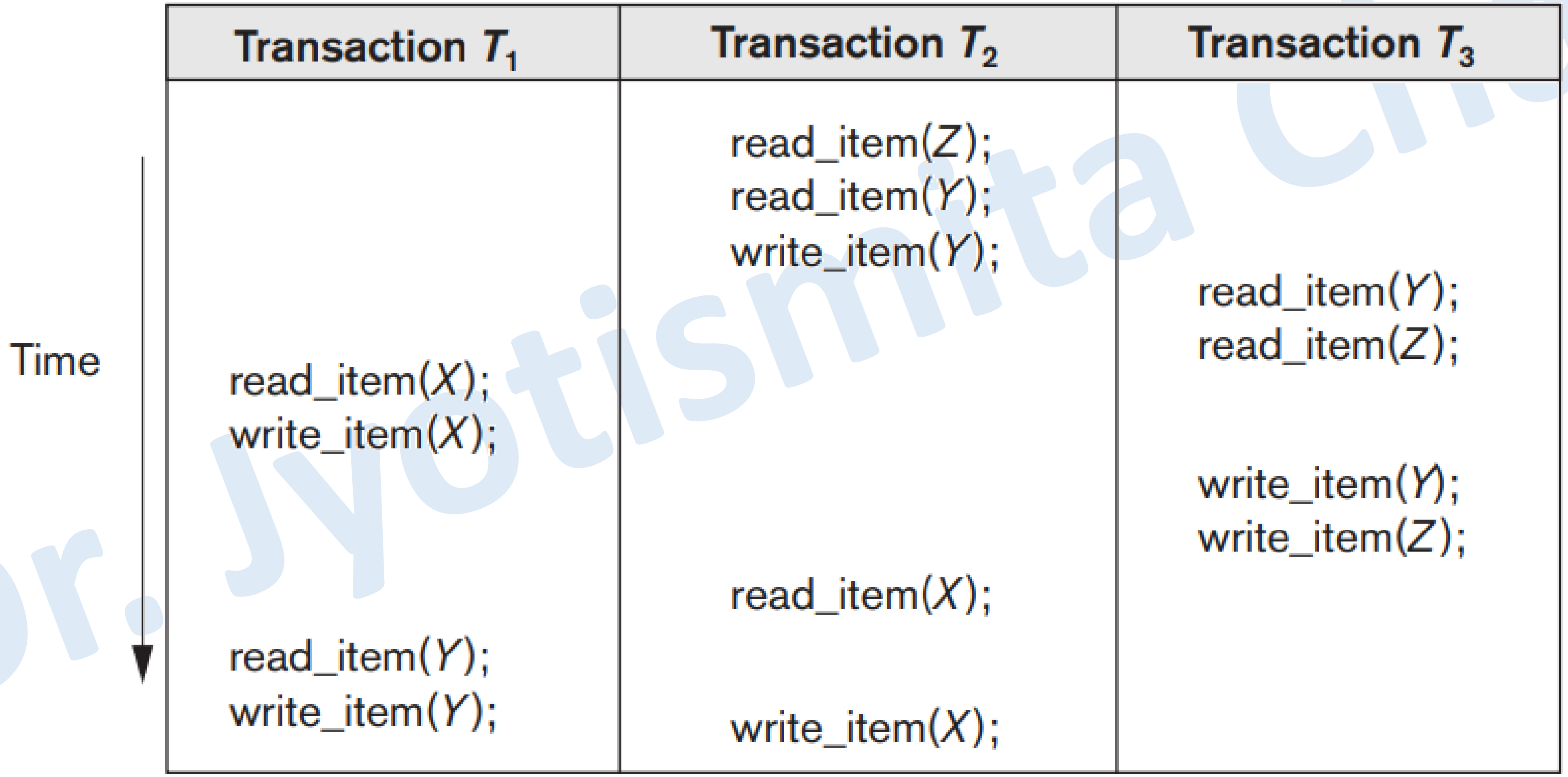
Transaction T_1
read_item(X);
write_item(X);
read_item(Y);
write_item(Y);

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

Transaction T_3
read_item(Y);
read_item(Z);
write_item(Y);
write_item(Z);

The read and write operations of three transactions T1, T2, and T3.

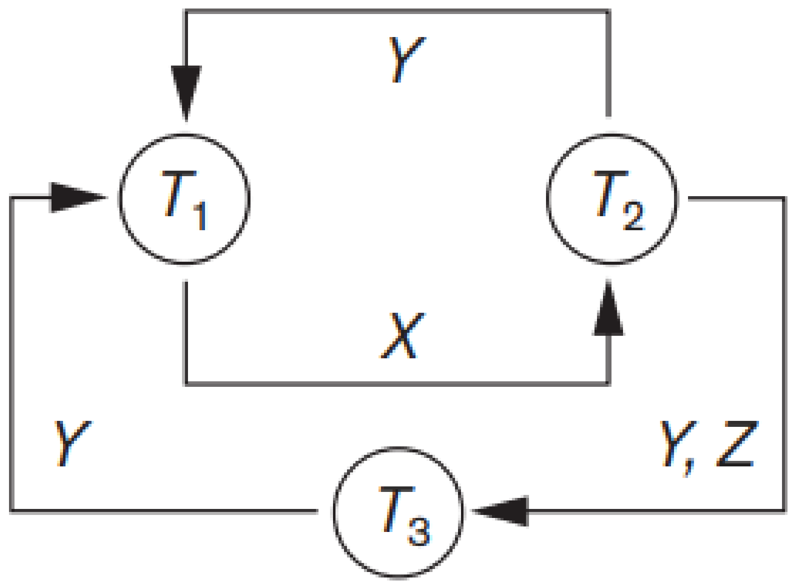
Testing for Serializability of a Schedule



Time	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

Testing for Serializability of a Schedule



Equivalent serial schedules

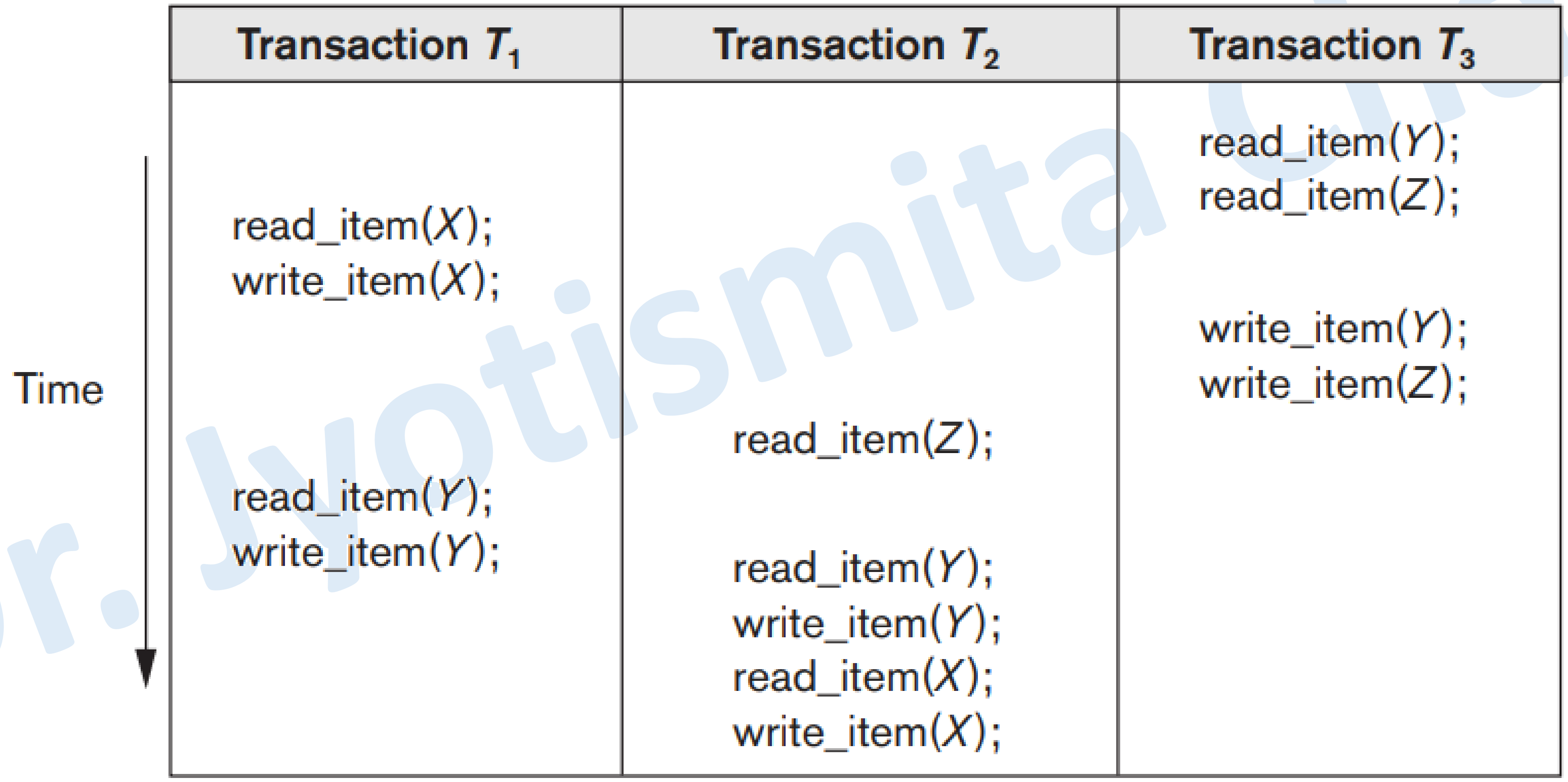
None

Reason

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

Precedence graph for schedule E.

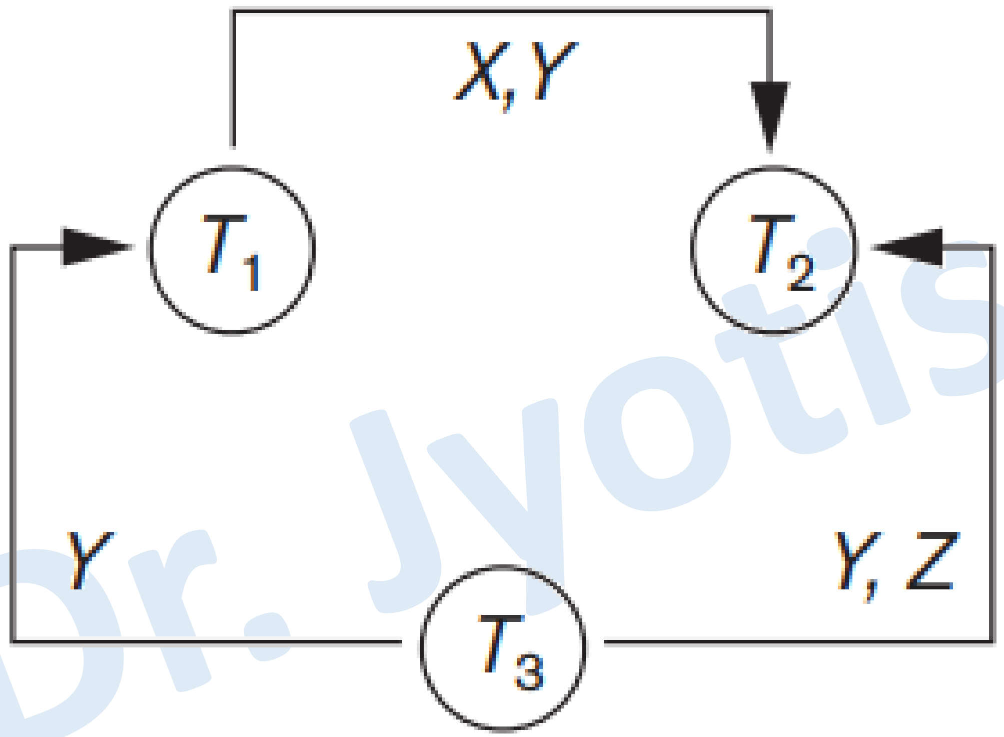
Testing for Serializability of a Schedule



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

Testing for Serializability of a Schedule

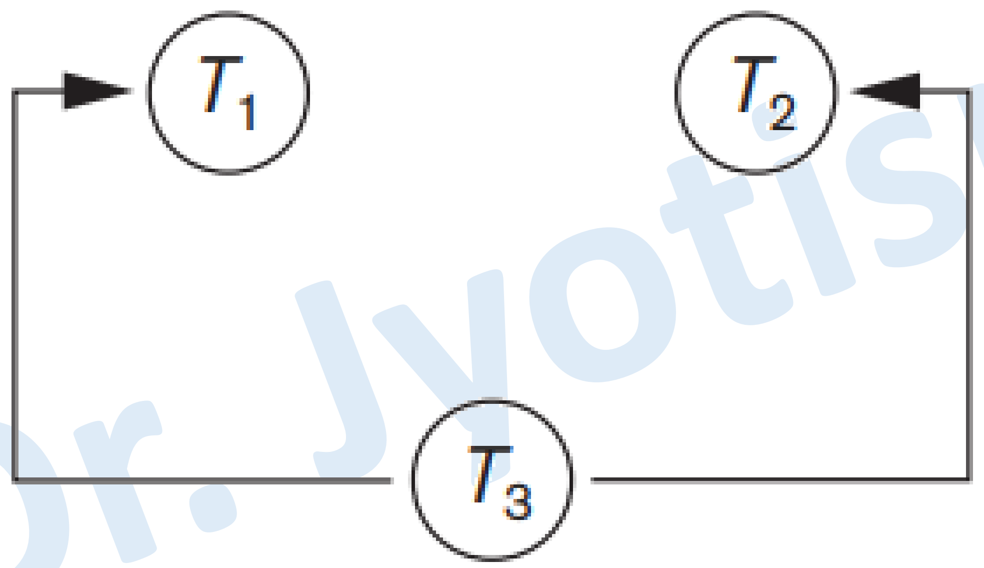


Equivalent serial schedules

$T_3 \rightarrow T_1 \rightarrow T_2$

Precedence graph for schedule F.

Testing for Serializability of a Schedule



Equivalent serial schedules

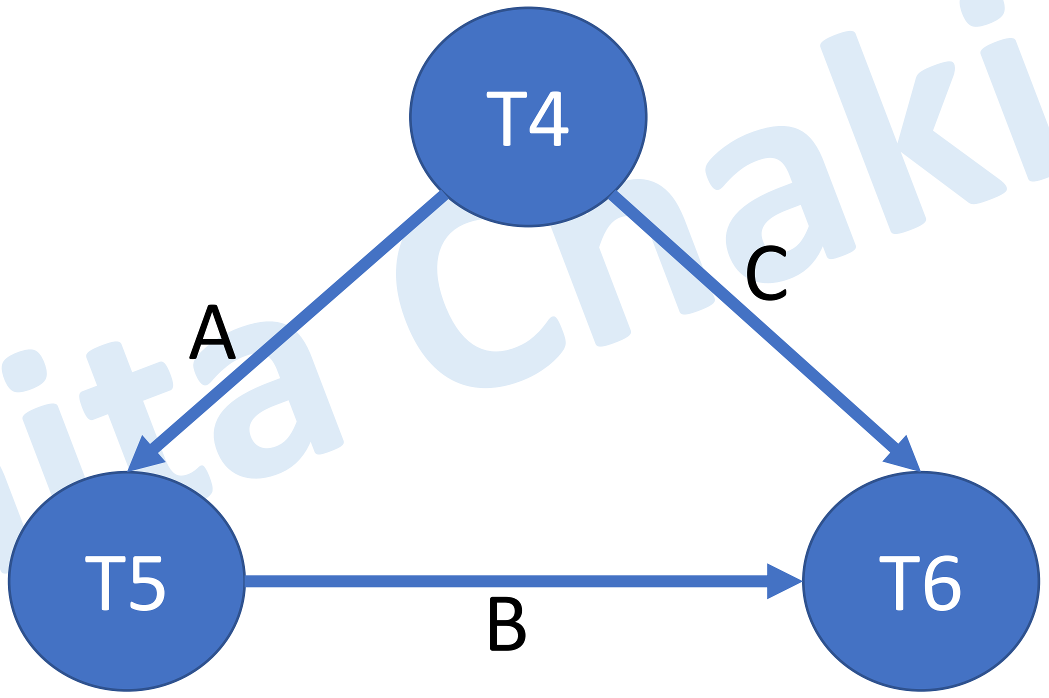
$T_3 \rightarrow T_1 \rightarrow T_2$

$T_3 \rightarrow T_2 \rightarrow T_1$

Precedence graph with two equivalent serial schedules

Testing for Serializability of a Schedule

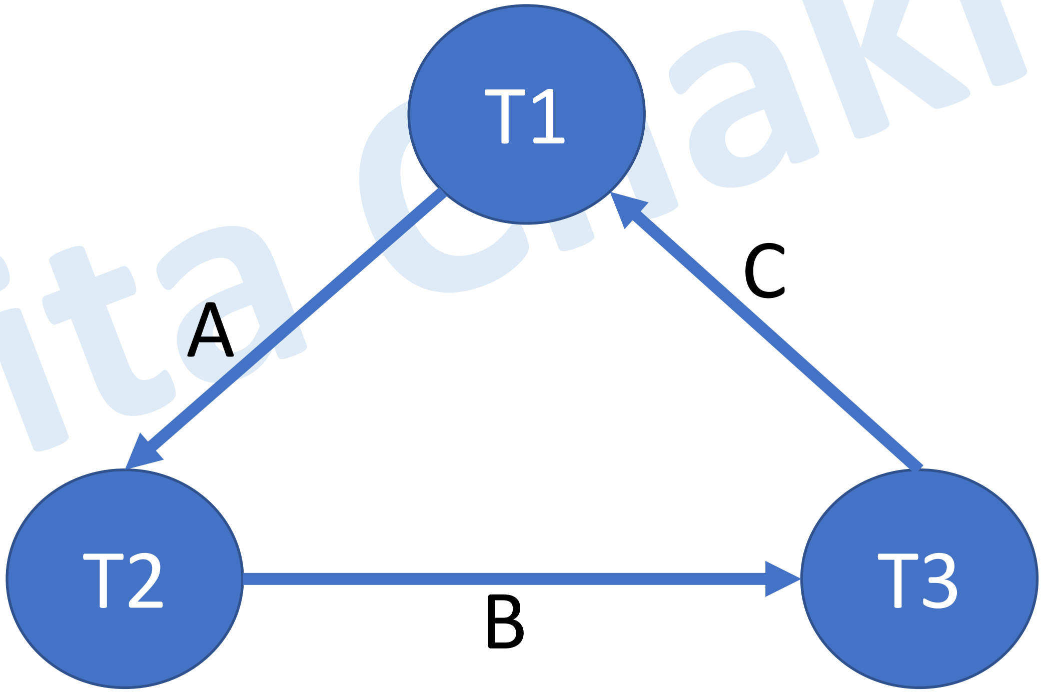
Time	T4	T5	T6
t1	Read (A)		
t2	A:= F1 (A)		
t3	Read (C)		
t4	Write (A)		
t5	A:= F2 (A)		
t6		Read (B)	
t7	Write (C)		
t8		Read (A)	
t9			Read (C)
t10		B:= F3 (B)	
t11		Write (B)	
t12			C:= F4 (C)
t13			Read (B)
t14			Write (C)
t15		A:= F5 (A)	
t16		Write (A)	
t17			B:= F6 (B)
t18			Write (B)



Equivalent serial schedule: T4 → T5 → T6

Testing for Serializability of a Schedule

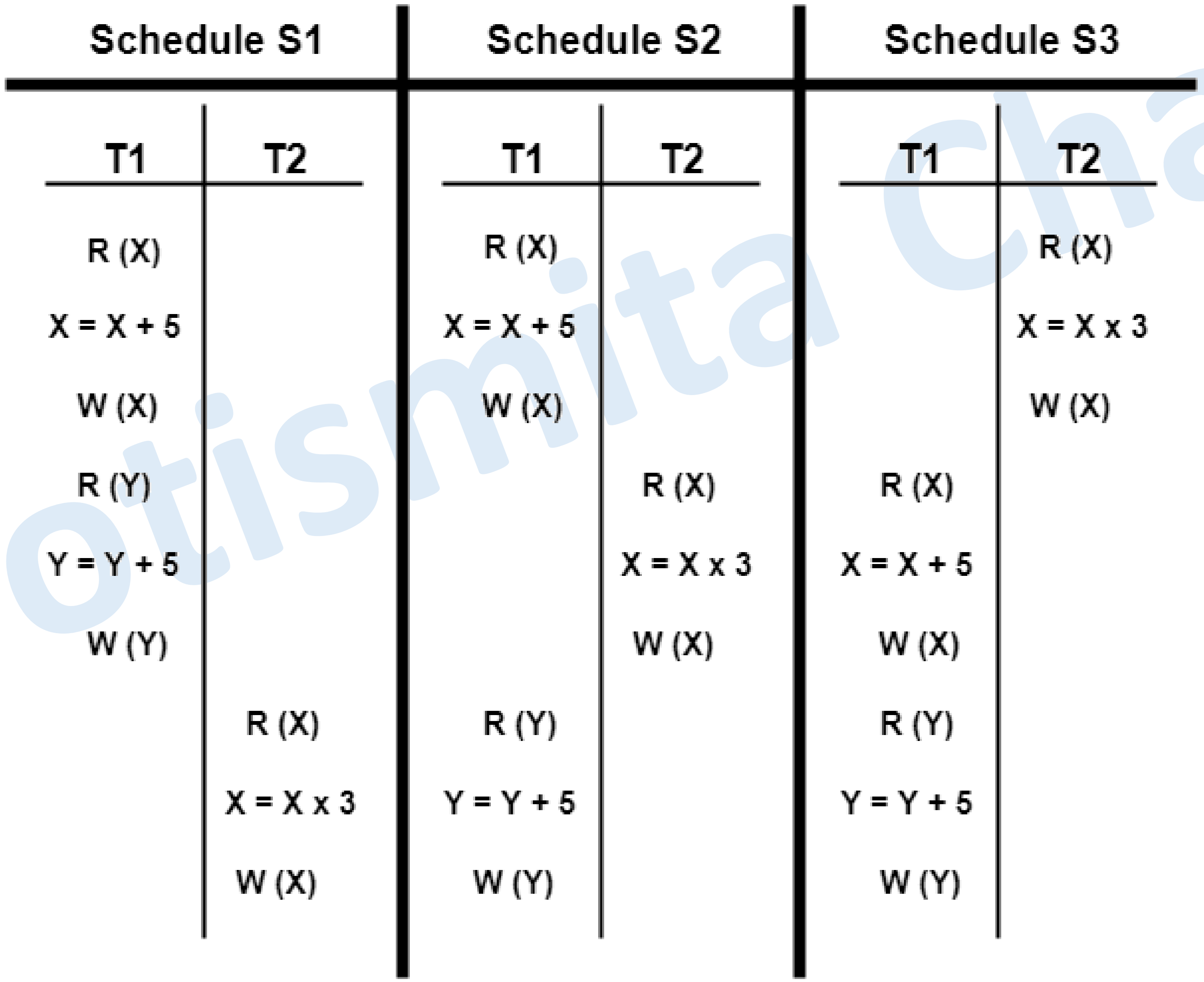
Time	T1	T2	T3
t1	Read (A)		
t2		Read (B)	
t3	A:= F1 (A)		
t4			Read (C)
t5		B:= F2 (B)	
t6		Write (B)	
t7			C:= F3 (C)
t8			Write (C)
t9	Write (A)		
t10			Read (B)
t11		Read (A)	
t12		A:= F4 (A)	
t13	Read (C)		
t14		Write (A)	
t15	C:= F5 (C)		
t16	Write (C)		
t17			B:= F6 (B)
t18			Write (B)



Equivalent serial schedule: NONE

Reason: Cycle A (T1 → T2), B (T2 → T3), C (T3 → T1)

Are the following three schedules result equivalent?



Schedule S1		Schedule S2		Schedule S3	
T1	T2	T1	T2	T1	T2
R (X)		R (X)			R (X)
X = X + 5		X = X + 5			X = X x 3
W (X)		W (X)			W (X)
R (Y)			R (X)	R (X)	
Y = Y + 5			X = X x 3	X = X + 5	
W (Y)			W (X)	W (X)	
	R (X)	R (Y)		R (Y)	
	X = X x 3	Y = Y + 5		Y = Y + 5	
	W (X)	W (Y)		W (Y)	

Solution

- To check whether the given schedules are result equivalent or not,
    - We will consider some arbitrary values of X and Y.
    - Then, we will compare the results produced by each schedule.
    - Those schedules which produce the same results will be result equivalent.
- Let X = 2 and Y = 5.
- On substituting these values, the results produced by each schedule are-
    - **Results by Schedule S1**- X = 21 and Y = 10
    - **Results by Schedule S2**- X = 21 and Y = 10
    - **Results by Schedule S3**- X = 11 and Y = 10
- Clearly, the results produced by schedules S1 and S2 are same.
- Thus, we conclude that S1 and S2 are result equivalent schedules.

Are the following two schedules conflict equivalent?

T1	T2
R (A)	
W (A)	
	R (A)
	W (A)
R (B)	
W (B)	

Schedule S1

T1	T2
R (A)	
W (A)	
R (B)	
W (B)	
	R (A)
	W (A)

Schedule S2

Solution

- To check whether the given schedules are conflict equivalent or not,
  - We will write their order of pairs of conflicting operations.
  - Then, we will compare the order of both the schedules.
  - If both the schedules are found to have the same order, then they will be conflict equivalent.
- **For schedule S1-**
  - The required order is-
    - $R_1(A)$, $W_2(A)$
    - $W_1(A)$, $R_2(A)$
    - $W_1(A)$, $W_2(A)$
- **For schedule S2-**
  - The required order is-
    - $R_1(A)$, $W_2(A)$
    - $W_1(A)$, $R_2(A)$
    - $W_1(A)$, $W_2(A)$
- Clearly, both the given schedules have the same order.
- Thus, we conclude that S1 and S2 are conflict equivalent schedules.