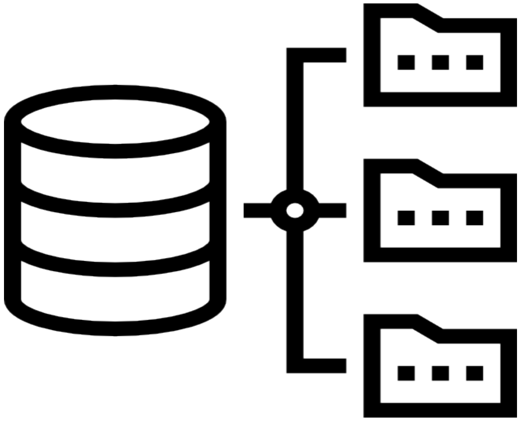
Advanced Database- IS411



Introduced by

Dr. Ebtsam Adel
Lecturer of Information Systems,
Information Systems department,
Faculty of computers and information,
Damanhour university

Lecture (02)

Chapter 20: Introduction to Transaction Processing Concepts and Theory

Part 2

Outline

- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability

Characterizing Schedules Based on Recoverability

Schedules of Transactions

- **Transaction schedule (history):** When transactions are executing concurrently in an interleaved fashion, then the **order of execution** of operations from the various transactions forms what is known as a **transaction schedule** (**history**).

- The following figure shows 4 possible schedules (A, B, C, D) of two transactions T1 and T2:
  - ✓ Order of operations from top to bottom
  - ✓ Each schedule includes same operations
  - ✓ Different order of operations in each schedule

Schedules of Transactions (cont.)

- Formal definition of a **schedule** (or **history**) **S** of n transactions T_1, T_2, ..., T_n is an **ordering** of all the operations of the transactions subject to the constraint that, for each transaction Ti that participates in 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.

Schedules of Transactions (cont.)

- Schedules can also be displayed in more compact notation.
- Order of operations from left to right.
- Include only read (r) and write (w) operations, with transaction id (1, 2, …) and item name (X, Y, …)
- Can also include other operations such as b (begin), e (end), c (commit), a (abort).

Schedules of Transactions (cont.)

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);	

Schedule C

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

Schedule D

Schedule C: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y);

Schedule D: r1(X); w1(X); r2(X); w2(X); r1(Y); w1(Y);

Schedules of Transactions (cont.)

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

Schedule A: r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(x);

Schedule B: r2(X); w2(X); r1(X); w1(X); r1(Y); w1(Y);

Schedules of Transactions- Conflict

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

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Check the conflict.

Conflict operations	
R1(x)	W2(x)
R2(x)	W1(x)
W1(x)	W2(x)

Schedules of Transactions- **complete schedule**

- A schedule S of n transactions T_1, T_2, … , T_n is said to be a **complete schedule** if the following conditions hold:

1. The operations in S are exactly those operations in T_1, T_2, … , T_n, including a commit or abort operation as the last operation for each transaction in the schedule.
2. For any pair of operations from the same transaction T_i, their relative order of appearance in S is the same as their order of appearance in T_i.
3. For any two conflicting operations, one of the two must occur before the other in the schedule.

Characterizing Schedules based on Recoverability

Characterizing Schedules based on Recoverability

- First, we would like to ensure that, 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 criteria are called **recoverable schedules**; Those do not do are called **nonrecoverable** and hence should not be **permitted** by the DBMS.

Characterizing Schedules based on Recoverability

Schedules classified into two main classes:

- **Recoverable schedule:** One where no committed transaction needs to be rolled back (aborted).

  A schedule S is **recoverable** if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

- **Non-recoverable schedule:** A schedule where a committed transaction may have to be rolled back during recovery.

Recoverable schedule: Write first must commit first.

Characterizing Schedules based on Recoverability

S_a': $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$

nonrecoverable

S_c: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$ nonrecoverable

S_d: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$ recoverable

S_e: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$ nonrecoverable

Characterizing Schedules Based on Recoverability (cont.)

- Schedule A below is **non-recoverable** because T2 reads the value of X that was written by T1, but then T2 commits before T1 commits or aborts
- To make it **recoverable**, the commit of T2 (c2) must be delayed until T1 either commits, or aborts (Schedule B).

- If T1 commits, T2 can commit
- If T1 aborts, T2 must also abort because it read a value that was written by T1; this value must be undone (reset to its old value) when T1 is aborted
  - known as **cascading rollback**

Characterizing Schedules based on Recoverability (cont.)

- 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.

- 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.

Characterizing Schedules based on Recoverability (cont.)

- 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.

Characterizing Schedules based on Recoverability (cont.)

- **Strict schedule:** A schedule in which a transaction T2 can neither read nor write an item X until the transaction T1 that last wrote X has committed.
- It is important to note that any strict schedule is also cascadeless, and any cascadeless schedule is also recoverable.

$$S_f: w_1(X, 5); w_2(X, 8); a_1;$$

Cascadeless but Unrestricted

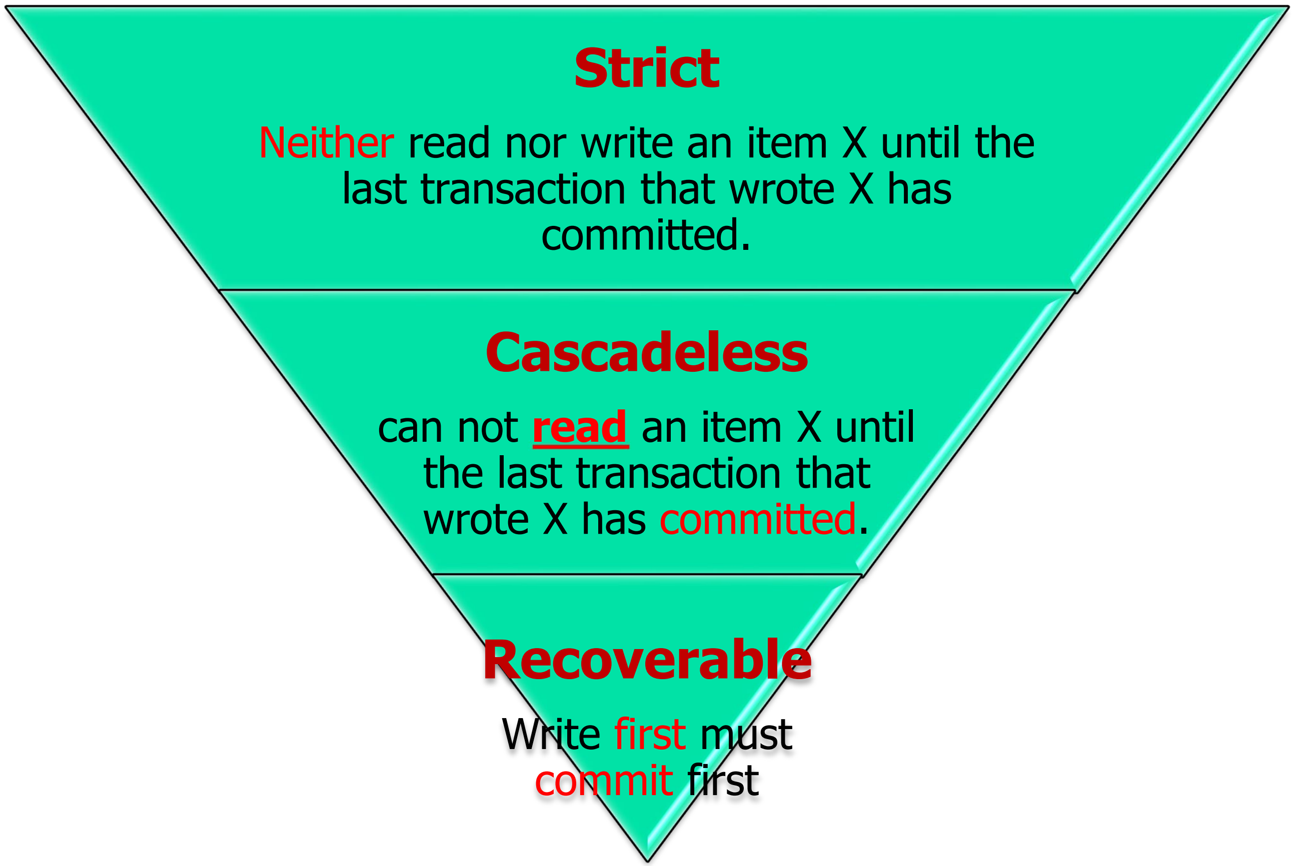
Example

X= 9
W1(X)= 5 BFIM=9
W2(X)= 8 BFIM=9
Abort T1;
X= 9

BFIM=Before image

Exercises

Exercises



- **Strict** "**means**" strict, cascadeless, recoverable
- **Cascadeless** "**means**" cascadeless, recoverable

Exercises

S_a: r1(x);r2(x);**w2(x)**;w1(x);**c2**;**r1(x)**;r1(y);c1 … **cascadeless**

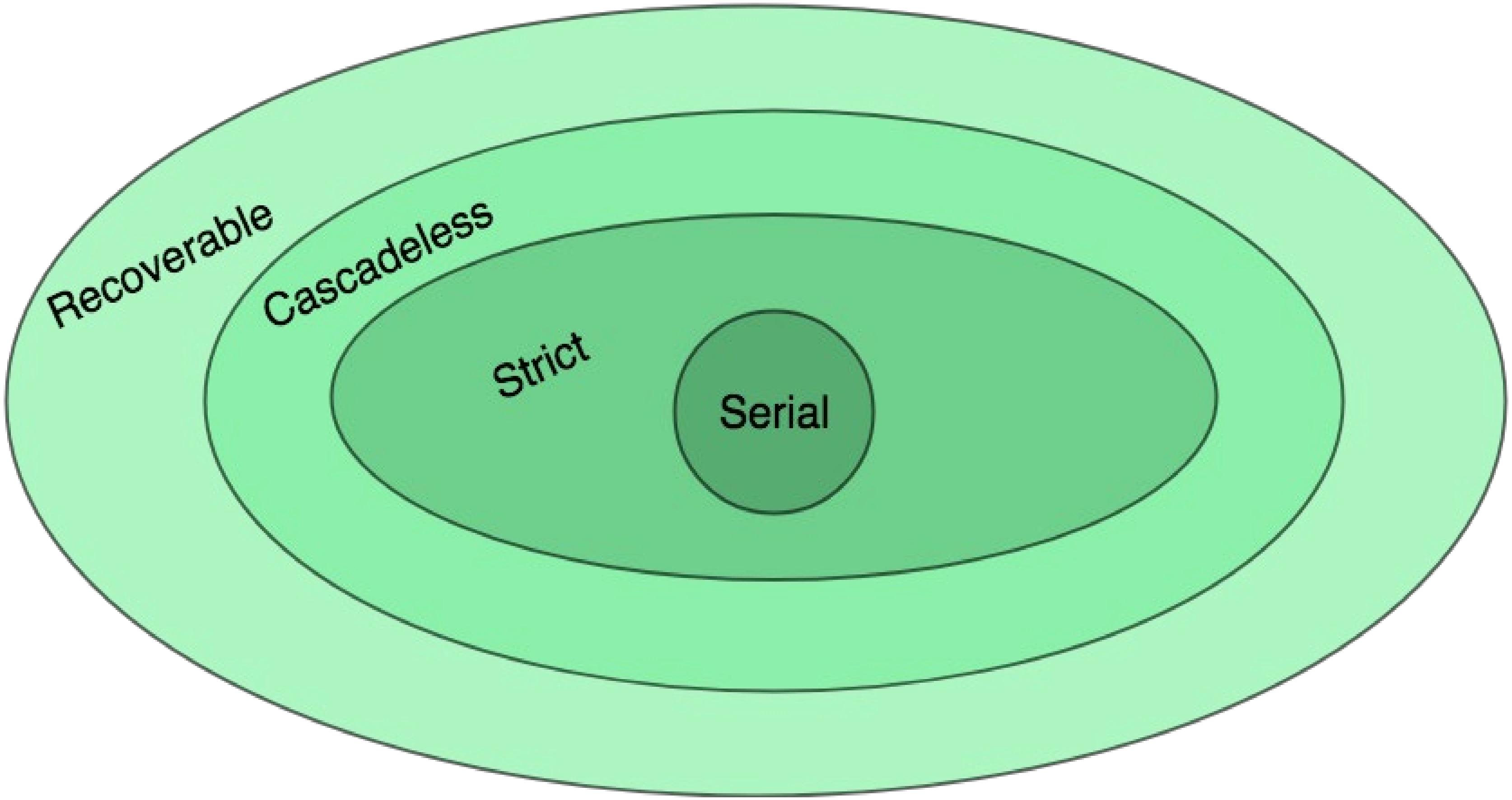
S_b: r1(x);r2(x);**w1(x)**;r1(y);w1(y);**c1,w2(x)**;c2…….**strict**

S_c: r1(x); **w1(x)**;r1(y);w1(y);**r2(x)**;**c1**;w2(x);**c2**…….**recoverable**

S_d: r1(x); **w1(x)**;r1(y);w1(y);**r2(x)**;w2(x);**c2**;**c1**…

non_recoverable

Types of schedules in DBMS



Outline

- Characterizing Schedules based on Recoverability
- Characterizing Schedules based on Serializability

Characterizing Schedules Based on Serializability

Characterizing Schedules based on Serializability (cont.)

20.5.1 Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figures 20.5(a) and (b) are called *serial* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: T_1 and then T_2 in Figure 20.5(a), and T_2 and then T_1 in Figure 20.5(b). Schedules C and D in Figure 20.5(c) are called *nonserial* because each sequence interleaves operations from the two transactions.

(a)

Time

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

(b)

Time

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

Two serial schedules

Characterizing Schedules based on Serializability (cont.)

(c)

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);	

Schedule C

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

Schedule D

Two non-serial schedules C and D with interleaving of operations.

Characterizing Schedules based on Serializability (cont.)

- **Serial schedule**: A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T are executed consecutively (without interleaving of operations from other transactions) in the schedule. **Otherwise**, the schedule is called **nonserial.**

- Based on the consistency preservation property, any serial schedule will produce a correct result (assuming no inter-dependencies among different transactions)

Characterizing Schedules based on Serializability

- The **problem** with serial schedules is that they **limit concurrency** by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus **wasting valuable** CPU **processing time**. Additionally, if some transaction T is long, the other transactions must wait for T to complete all its operations before starting. Hence, serial schedules are **unacceptable** in practice.

- However, if we can determine which other schedules are equivalent to a serial schedule, we can allow these schedules to occur.

Characterizing Schedules based on Serializability (cont.)

- Serial schedules are not feasible for performance reasons:
  - No interleaving of operations
  - Long transactions force other transactions to wait
  - System cannot switch to other transaction when a transaction is waiting for disk I/O or any other event.
  - Need to allow concurrency with interleaving without sacrificing correctness

Characterizing Schedules based on Serializability (cont.)

- **Serializable schedule (قابل للتسلسل)**: A schedule S is **serializable** if it is **equivalent** to some **serial** schedule of the same n transactions.

- There are (n)! serial schedules for n transactions – a serializable schedule can be equivalent to some serial schedule of the same transactions.

- **Question:** How do we define equivalence of schedules?

a. Result equivalent

b. Conflict equivalent

Characterizing Schedules based on Serializability (cont.)

- **Serializable schedule**: A schedule S is **serializable** if it is **equivalent** to some serial schedule of the same n transactions.

Question: How do we define **equivalence** of schedules?

a. Result equivalent
b. Conflict equivalent

Equivalence of Schedules- **Result equivalent**

- There are several ways to define schedule equivalence. The simplest but least satisfactory definition involves comparing the effects of the schedules on the database.

- **Result equivalent**: Two schedules are called result equivalent if they produce the **same final state** of the database.

Equivalence of Schedules- **Result equivalent**

Figure 21.6
Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

S_1	S_2

read_item(X);
$X := X + 10$;
write_item(X);
	read_item(X);
	$X := X * 1.1$;
	write_item (X);

Two schedules are the same result when x=100 **only**.

Equivalence of Schedules- **Result equivalent**

- Difficult to determine without analyzing the internal operations of the transactions, which is not feasible in general.

- May also get result equivalence by chance for a particular input parameter even though schedules are not equivalent in general.

Equivalence of Schedules - **Conflict equivalent**

- **Conflict equivalent**: Two schedules are conflict equivalent if the order of any two conflicting operations is the same in both schedules.

- Commonly used definition of schedule equivalence
- Two operations are **conflicting** if:
  - They access the **same data item** X
  - They are from two **different transactions**
  - At least one is a **write operation**
- Read-Write conflict example: r1(X) and w2(X)
- Write-write conflict example: w1(Y) and w2(Y)

Equivalence of Schedules - **Conflict equivalent**

S1

T1	T2

W(X)	
	R(X)

S2

T1	T2
W(X)	
	R(X)

Conflict equivalent

S1

T1	T2

W(X)	
	R(X)

S2

T1	T2

R(X)	
	W(X)

Not Conflict equivalent

Equivalence of Schedules (cont.)

- Changing the order of conflicting operations generally causes a different outcome.

- **Example:** changing r1(X); w2(X) to w2(X); r1(X) means that T1 will read a different value for X
- **Example:** changing w1(Y); w2(Y) to w2(Y); w1(Y) means that the final value for Y in the database can be different
- Note that read operations are **not conflicting**; changing r1(Z); r2(Z) to r2(Z); r1(Z) does not change the outcome.

Conflict serializable schedule

- Using the notion of conflict equivalence, we define a schedule *S* to be **serializable** if it is (**conflict**) equivalent to some serial schedule *S'*.

- **In such a case**, we can **reorder** the **non-conflicting** operations in *S* until we form the **equivalent serial** schedule *S'*.

Conflict serializable schedule

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

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

Schedule A=schedule D
Conflict serializable schedule

Conflict serializable schedule

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

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

Schedule A

Not-Conflict serializable schedule

Testing for Serializability of a Schedule

Testing for Serializability of a Schedule

- There is a simple algorithm for determining whether a particular schedule is (conflict) serializable or not.

- Most concurrency control methods do *not* actually test for Serializability. Rather protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable.

- Some methods guarantee Serializability in most cases, but do not guarantee it absolutely, in order to reduce the overhead of concurrency control.

Testing for Serializability of a Schedule

Algorithm 20.1 can be used to test a schedule for conflict serializability. 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 $N = \{T_1, T_2, \ldots, T_n\}$ and a set of directed edges $E = \{e_1, e_2, \ldots, e_m\}$. 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 $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \leq n$, where T_j is the **starting node** of e_i and T_k is the **ending node** of ei. Such an edge from node T_j to node T_k is created by the algorithm if a pair of conflicting operations exist in T_j and T_k and the conflicting operation in T_j appears in the schedule *before* the *conflicting operation* in T_k.

Testing for Serializability of a Schedule

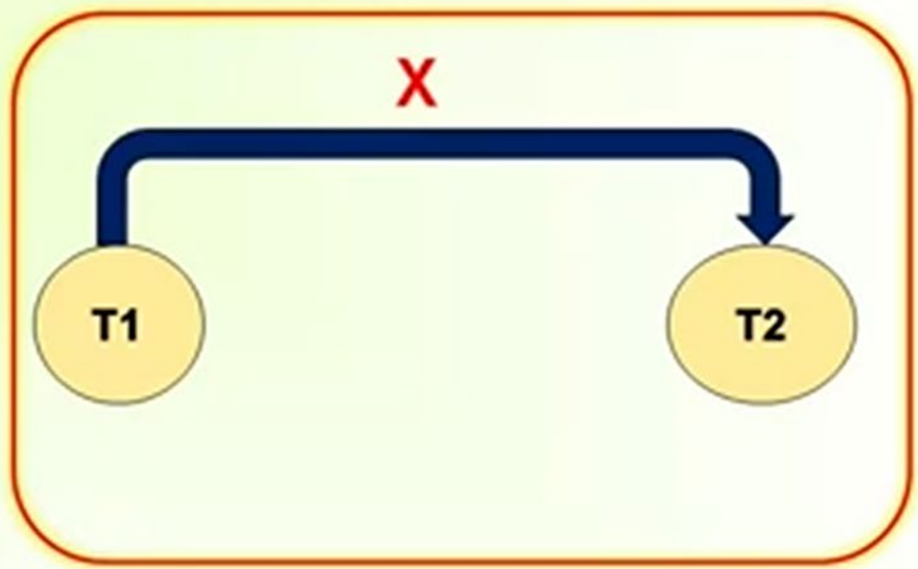
Algorithm 20.1. Testing Conflict Serializability of a Schedule S

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
3. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
4. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

Testing for Serializability of a Schedule

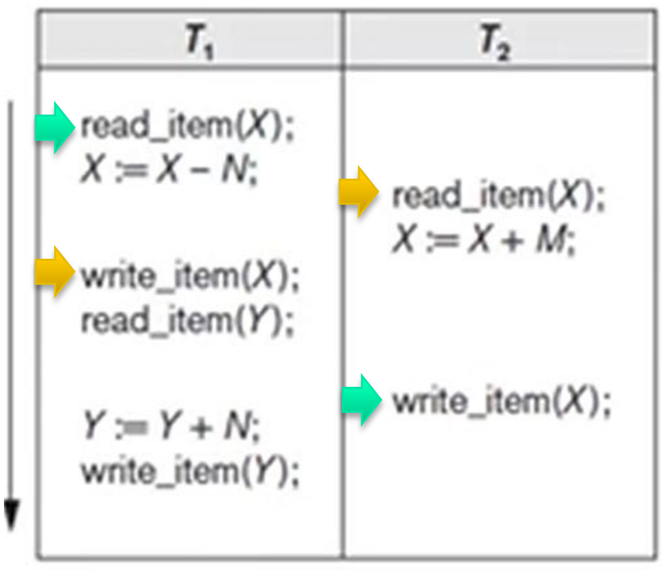
T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

Schedule A

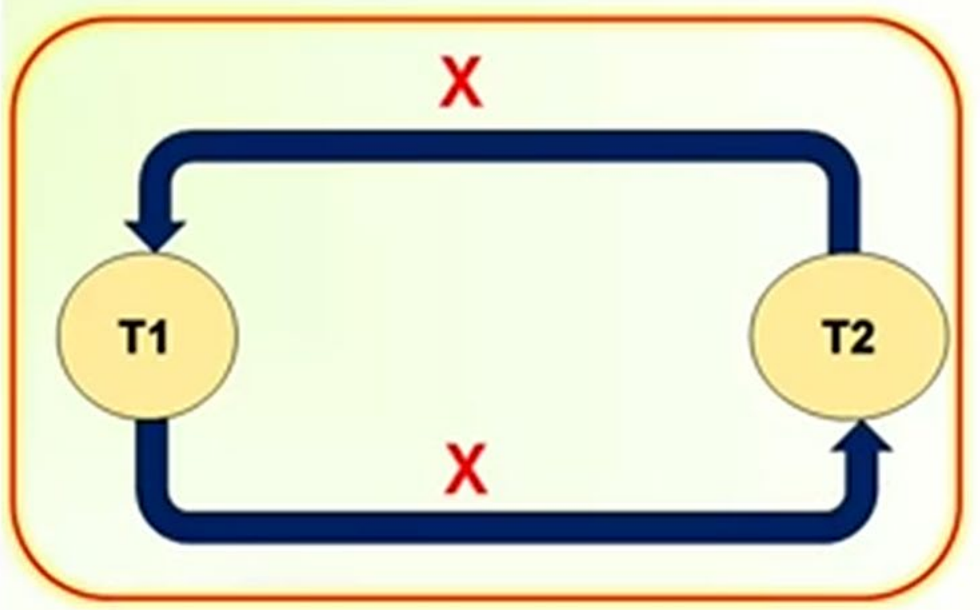


No cycles,
Serializable schedule

Testing for Serializability of a Schedule

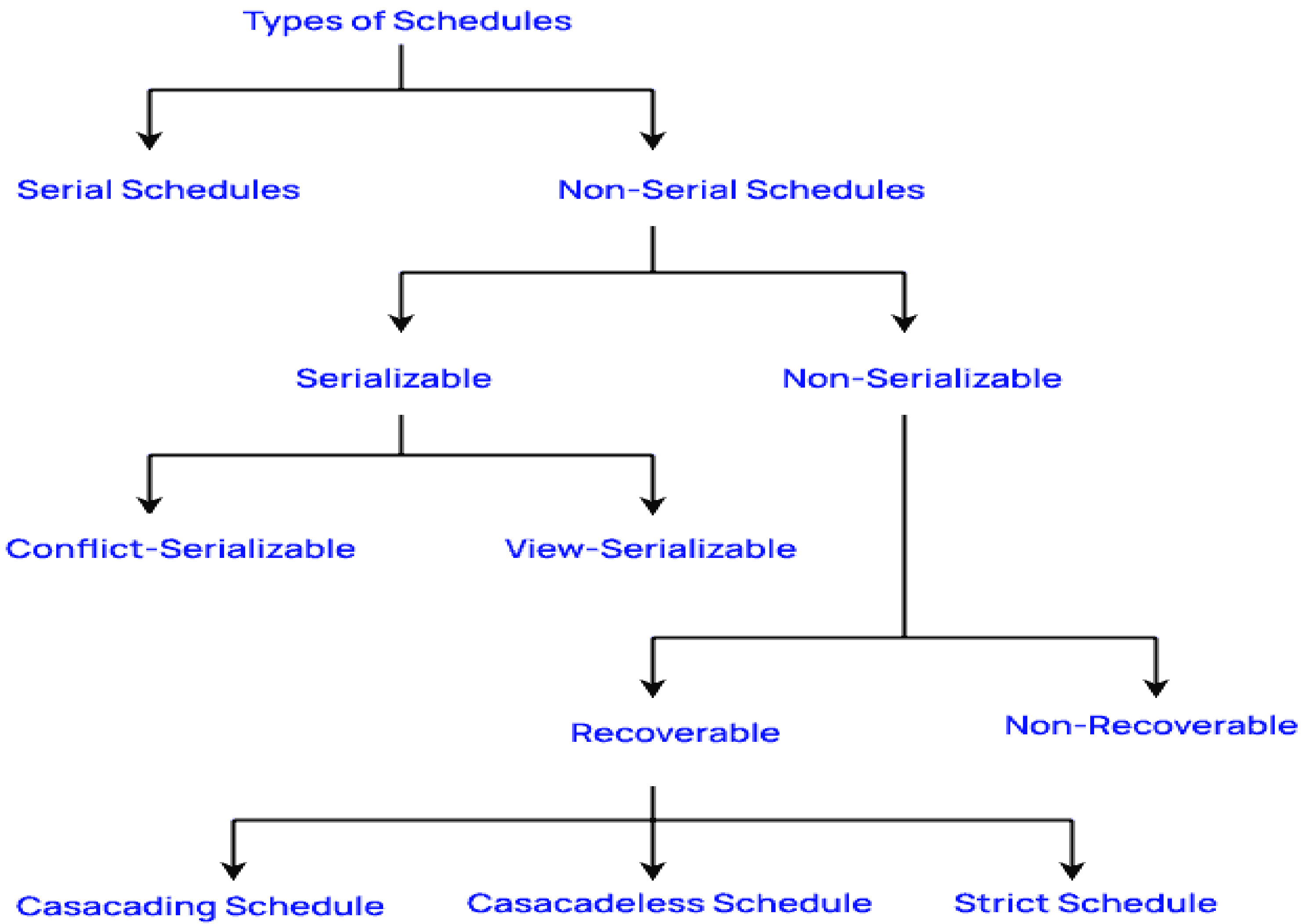


Schedule C



Has cycles, in the same direction
Non serializable

Types of schedules in DBMS




Thank You!
Any questions?