Lesson 9 - Chapter 22

Transaction Management

Serializability and Recoverability

Life without problems

WHOLENESS OF THE LESSON

A transaction has the qualities of atomicity (all or none), consistency (transaction running alone preserves the consistency of the database), isolation (does not interfere with others), and durability (once complete, changes survive even system failures). Atomicity and durability are achieved by recovery mechanisms and isolation by concurrency control. Science & Technology of Consciousness: The qualities of the unified field include reverberating wholeness, all knowingness, integrating and harmonizing, invincible, and progressive. These qualities are enlivened by contact of the mind with the unified field of all the laws of nature during the TM technique.

Transaction Support

- Transaction is an **action**, or **series of actions**, carried out by a **single user** or **application program**, that **reads** or **updates** the contents of the **database**.
- A transaction is treated as a **logical unit of work** on the DB. It may be an entire program, a part of a program, or a single statement (INSERT or UPDATE) and it may involve any number of operations on the DB.
- Application program is series of transactions with non-database processing in between.
- A transaction should always transform the database from one consistent state to another, although consistency may be violated during transaction.

Examples of Transaction

Staff (<u>staffNo</u>, fName, lName, position, sex, DOB, salary, branchNo)

PropertyForRent (<u>propertyNo</u>, street, city, postcode, type, rooms, rent, ownerNo, staffNo, branchNo)

```
read(staffNo = x, salary)
salary = salary * 1.1
write(staffNo = x, salary)
```

Database Operation

Non Database Operation

```
delete(staffNo = x)
for all PropertyForRent records, pno
begin
    read(propertyNo = pno, staffNo)
    if (staffNo = x) then
    begin
        staffNo = newStaffNo
        write(propertyNo = pno, staffNo)
    end
end
```

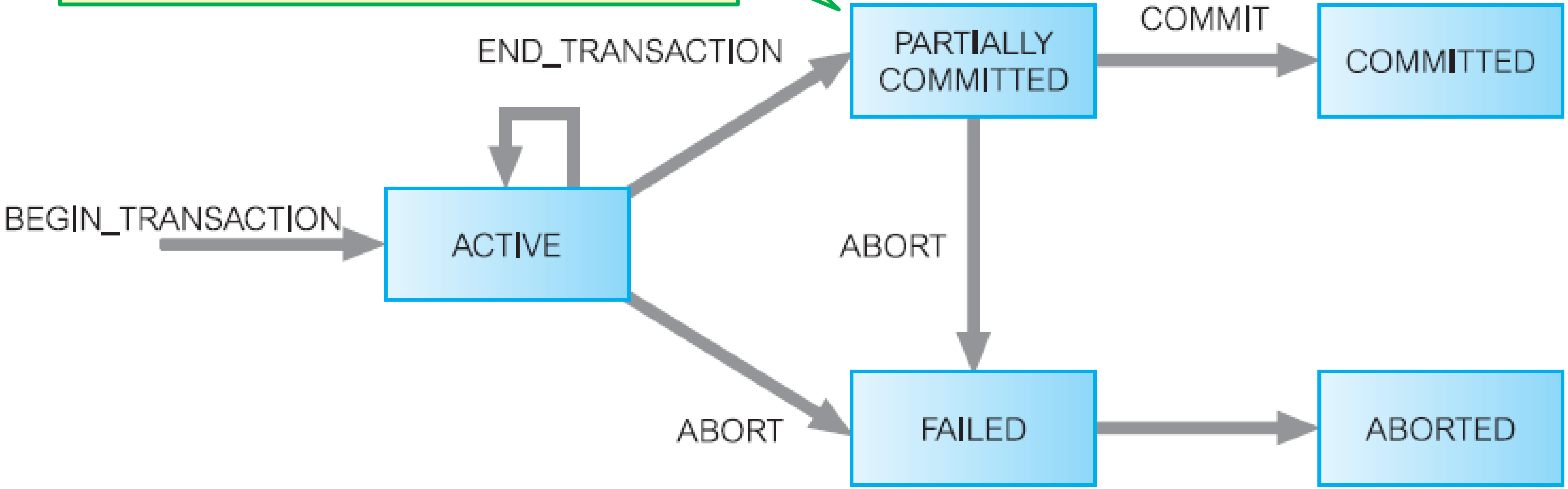
Transaction Support contd..

- **Transaction can have one of two outcomes:**
  - **Success** - transaction *commits* and database reaches a new consistent state.
  - **Failure** - transaction *aborts (ends)*, and database must be restored back to consistent state before it started.
    - » Such a transaction is *rolled back* or *undone*.
- Committed transaction cannot be aborted.
- Aborted transaction that is rolled back can be restarted later.
- For delimiting transactions, the following keywords are used in many DMLs.

BEGIN TRANSACTION, COMMIT, ROLLBACK

State Transition Diagram for a Transaction

This state occurs after the final statement has been executed. At this point, it may be found that the transaction has violated an integrity constraint or serializability, and the transaction needs to be aborted.



Properties of Transactions - ACID

- A transaction is a very small unit of a program and it may contain several low-level tasks.
- A transaction in a database system must maintain Atomicity, Consistency, Isolation, and Durability − commonly known as **ACID properties** − in order to ensure accuracy, completeness, and data integrity.

Atomicity	'All or nothing' property.
Consistency	A transaction must transform database from one consistent state to another.
Isolation	Partial effects of incomplete transactions should not be visible to other transactions. (Transactions execute independently of one another.)
Durability	Effects of a committed transaction are permanent and must not be lost because of later failure.

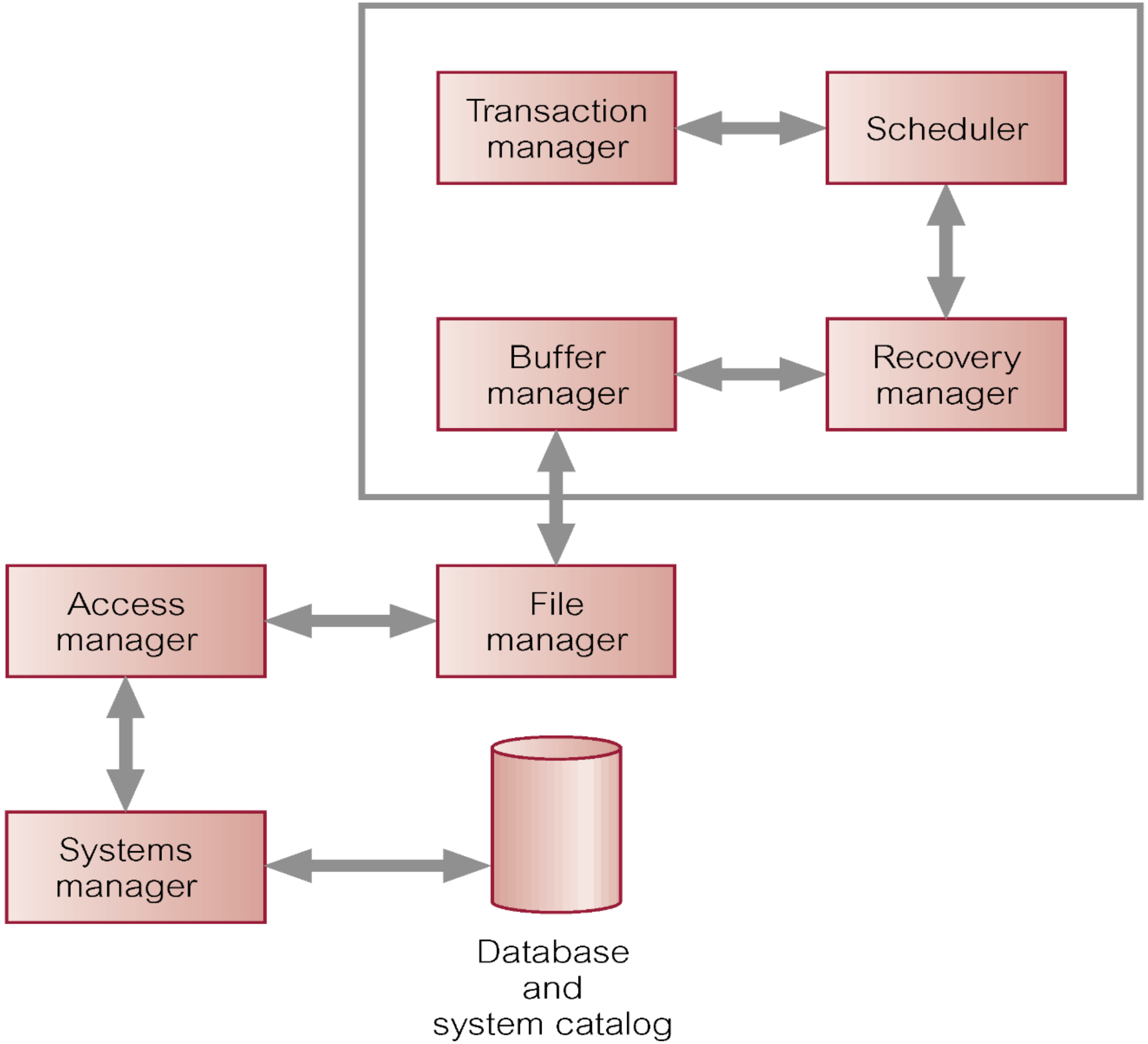
Properties of Transactions contd..

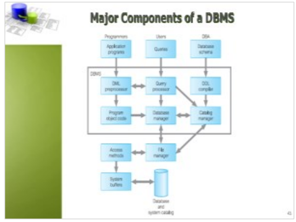
- **Atomicity** → A transaction is an indivisible unit that is either performed in its entirety or is not performed at all.
  - It is the responsibility of the recovery subsystem of the DBMS to ensure atomicity.
- **Consistency** → It is the responsibility of both the DBMS and the application developers to ensure consistency.
  - The DBMS can ensure consistency by enforcing all the constraints that have been specified on the database schema, such as integrity constraints. However, in itself this is insufficient to ensure consistency.
    - E.g., suppose that we have a transaction that is intended to transfer money from one bank account to another and the programmer makes an error in the transaction logic and debits one account but credits the wrong account; then the database is in an inconsistent state. However, the DBMS would not have been responsible for introducing this inconsistency and would have had no ability to detect the error.

Properties of Transactions contd..

- **Isolation** → Transactions execute independently of one another. In other words, the partial effects of incomplete transactions should not be visible to other transactions.
  - It is the responsibility of the concurrency control subsystem to ensure isolation.
- **Durability** → The effects of a successfully completed (committed) transaction are permanently recorded in the database and must not be lost because of a subsequent failure.
  - It is the responsibility of the recovery subsystem to ensure durability.

DBMS Transaction Subsystem





Concurrency Control

- A major objective in developing a DBMS is to enable many users to access shared data concurrently. In this section we examine the problems that can arise with concurrent access and the techniques that can be employed to avoid these problems.
- **Concurrency Control :- Process of managing simultaneous operations on the database without having them interfere with one another.**
- Prevents interference when two or more users are accessing database simultaneously and at least one is updating data.
- Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Need for Concurrency Control

- Three examples of potential problems caused by concurrency:
  - **Lost update problem**
  - **Uncommitted dependency problem**
  - **Inconsistent analysis problem**

Lost Update Problem

- **An apparently successfully completed update operation by one user can be overridden by another user.**
- T_2 withdrawing \$10 from an account with bal_x initially \$100.
- T_1 depositing \$100 into the same account.
- Serially, final balance would be \$190 no matter which transaction is performed first.

Time	T_1	T_2	bal_x
t_1	begin_transaction		100
t_2	read(bal_x)	begin_transaction	100
t_3	$bal_x = bal_x + 100$	read(bal_x)	100
t_4	write(bal_x)	$bal_x = bal_x - 10$	200
t_5	commit	write(bal_x)	90
t_6		commit	90

Loss of T_1's update can be avoided by preventing T_2 from reading bal_x until after the update is committed by T_1.

Uncommitted Dependency Problem
(Dirty Read Problem)

- **Occurs when one transaction is allowed to see the intermediate results of another transaction before it has committed.**
- T_3 updates bal_x to \$200 but it aborts, so bal_x should be back at original value of \$100.
- T_4 has read new value of bal_x (\$200) and uses that value as a basis for \$10 reduction, giving a new balance of \$190, instead of \$90.

Uncommitted Dependency Problem (Dirty Read Problem) contd..

Time	T_3	T_4	bal_x
t_1	begin_transaction		100
t_2	read(bal_x)		100
t_3	$bal_x = bal_x + 100$		100
t_4	write(bal_x)	begin_transaction	200
t_5	⋮	read(bal_x)	200
t_6	rollback	$bal_x = bal_x - 10$	100
t_7		write(bal_x)	190
t_8		commit	190

Problem can be avoided by preventing T_4 from reading bal_x until after T_3 commits or aborts.

Inconsistent Analysis Problem

- **Occurs when transaction reads several values but second transaction updates some of them during execution of first.**
- T_5 is totaling balances of account x ($100), account y ($50), and account z ($25).
- Meantime, T_6 has transferred $10 from balx to balz, so T_5 now has wrong result ($10 too high).

Inconsistent Analysis Problem

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1	begin_transaction		100	50	25	
t_2	sum = 0	begin_transaction	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	sum = sum + bal_x	$bal_x = bal_x - 10$	100	50	25	100
t_5	read(bal_y)	write(bal_x)	90	50	25	100
t_6	sum = sum + bal_y	read(bal_z)	90	50	25	150
t_7		$bal_z = bal_z + 10$	90	50	25	150
t_8		write(bal_z)	90	50	35	150
t_9	read(bal_z)	commit	90	50	35	150
t_{10}	sum = sum + bal_z		90	50	35	185
t_{11}	commit		90	50	35	185

Problem can be avoided by preventing T_5 from reading bal_x and bal_z until after T_6 completed updates.

Concurrency Control Protocol : Serializability

- Objective of a concurrency control protocol is to schedule transactions in such a way so as to avoid any interference between them and hence prevent the types of problems described earlier.
- Simple solution is to run transactions serially, but this limits the degree of concurrency or parallelism in system.
- Serializability identifies those executions of transactions guaranteed to ensure consistency.

What is a Schedule?

Schedule :- A sequence of operations (read, write, commit, rollback) by a set of concurrent transactions that **preserves** the **order of operations** in each of the individual transactions.

(T1, R(x)), (T1, W(x))

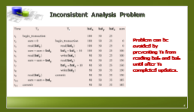
(T2, R(y)), (T2, W(y))

Transactions T1 and T2

Ex.1 → (T1, R(x)), (T1, W(x)), (T2, R(y)), (T2, W(y))

Ex.2 → (T1, R(x)), (T2, R(y)), (T1, W(x)), (T2, W(y))

Ex.3 → (T1, R(x)), (T2, R(y)), (T2, W(y)), (T1, W(x))



Serial Schedule

Serial Schedule :- A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.

(T1, R(x)), (T1, W(x))

(T2, R(y)), (T2, W(y))

Transactions T1 and T2

(T1, R(x)), (T1, W(x)), (T2, R(y)), (T2, W(y)) // T1, T2

(T2, R(y)), (T2, W(y)), (T1, R(x)), (T1, W(x)) // T2, T1

Nonserial Schedule

A schedule where the operations from a set of concurrent transactions are interleaved.

(T1, R(x)), (T1, W(x))
(T2, R(y)), (T2, W(y))
$\Big\}$ Transactions T1 and T2

// neither T1,T2 nor T2, T1

(T1, R(x)), (T2, R(y)), (T2, W(y)), (T1, W(x))

(T1, R(x)), (T2, R(y)), (T1, W(x)), (T2, W(y))

(T2, R(y)), (T1, R(x)), (T1, W(x)), (T2, W(y))

(T2, R(y)), (T1, R(x)), (T2, W(y)), (T1, W(x))

Serializability

- Two serial schedules need not produce the same result. That means there is no guarantee that results of all serial executions of a given set of transactions will be identical.
- However, a serial schedule will always leave the database in a consistent state. Therefore, every serial execution is considered correct.
- The **objective of serializability** is to **find nonserial schedules** to execute concurrently **without interfering with one other**, and thereby produce a database state that **could be** produced by a serial schedule.
- In other words, we want to find nonserial schedules that are equivalent to some serial schedule. Such a schedule is called **serializable schedule**.

Serializable Schedule

- If a set of transactions execute concurrently, we say that this *nonserial* schedule is a **Serializable schedule** if it produces the same results as some serial execution regardless of the specific information in the database.
- Definition of serializability is a bit difficult to handle: How can we test for the same effect regardless of data?
- To come up with an answer, we'll create a stricter definition of serializability, called ***conflict-serializability*.**

Conflict

In Serializability, the ordering of read and write operations are important.

a) If two transactions only read a data item then there is *no conflict* and order is not important.

b) If two transactions either read or write separate data items then there is *no conflict* and order is not important.

c) If one transaction writes a data item and another transaction either reads or writes the *same data item*, then there is a ***conflict*** and order of execution of these transactions is very important.

Conflict Examples

$(T_i, R(x)), (T_j, W(x))$	// conflict
$(T_i, W(x)), (T_j, R(x))$	// conflict
$(T_i, W(x)), (T_j, W(x))$	// conflict
$(T_i, R(x)), (T_j, W(y))$	// No conflict
$(T_i, W(x)), (T_j, W(y))$	// No conflict
$(T_i, R(x)), (T_j, R(x))$	// No conflict
$(T_i, W(x)), (T_j, R(y))$	// No conflict

Conflict-equivalence

- Two schedules are **conflict-equivalent** if one can be obtained from the other through a series of **swaps** of adjacent operations.
  - Swapping between operations is allowed only in a way such that the final result will remain the same.

No swap for the following patterns:

- If the adjacent operations in the schedule are of the same transaction, then DO NOT swap. (why?)
- If the adjacent operations use the same database element, and at least one of the operations is a write, then DO NOT swap. (why?)

Conflict-equivalence

Example (Swapping) –

Given a non serial schedule S as follows:

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

Find a conflict equivalent serial schedule for S.

Conflict-equivalence

Given non serial schedule S as:

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

Start Swapping:

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

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

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

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

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

T2, T1 ➡ Final equivalent serial schedule

Conflict Serializability

- A schedule is **conflict-serializable** if it is conflict-equivalent to some serial schedule.
- A conflict serializable schedule orders any conflicting operations in the same way as some serial execution.

Example: Conflict-Serializable Schedule

- Consider the nonserial schedule S:

  (T1, R(X)), (T2, R(Y)), (T3, W(X)), (T2, R(X)), (T1, R(Y))

- S is conflict-equivalent to the following schedule:

  (T1, R(X)), (T1, R(Y)), (T3, W(X)), (T2, R(Y)), (T2, R(X))

- Thus S is conflict-equivalent to the serial schedule T1, T3, T2.

- **So, S is a conflict serializable schedule.**
  - Meaning, it'll keep the database in consistent state even after executing all the concurrent operations as per the given sequence!

Testing for Conflict Serializability

An alternate way to determine whether a nonserial schedule S is conflict serializable is to create a precedence graph.

Precedence (Serialization) Graph :

- A node for each transaction
- A directed edge (Ti → Tj) if Ti **writes** a value before Tj **reads/writes** it
- A directed edge (Ti → Tj) if Ti **reads** a value before Tj **writes** it

If the precedence graph contains a cycle, then the schedule is **not-conflict serializable**.

Precedence Graph

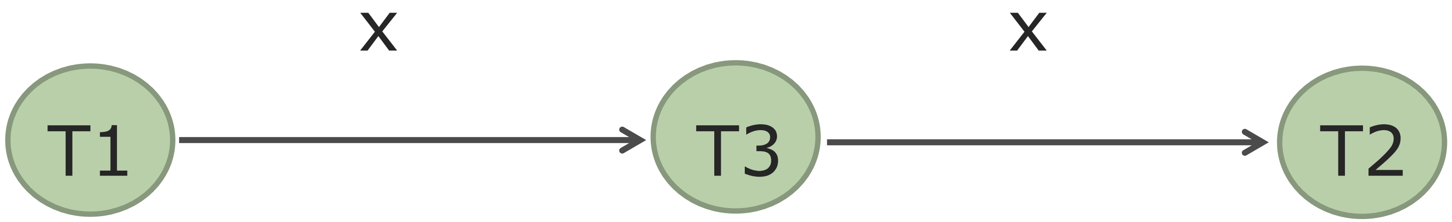
A directed edge (Ti → Tj) if Ti **writes** a value before Tj **reads/writes** it

A directed edge (Ti → Tj) if Ti **reads** a value before Tj **writes** it

- Consider the nonserial schedule S :

  (T1, R(X)), (T2, R(Y)), (T3, W(X)), (T2, R(X)), (T1, R(Y))

- Precedence Graph:

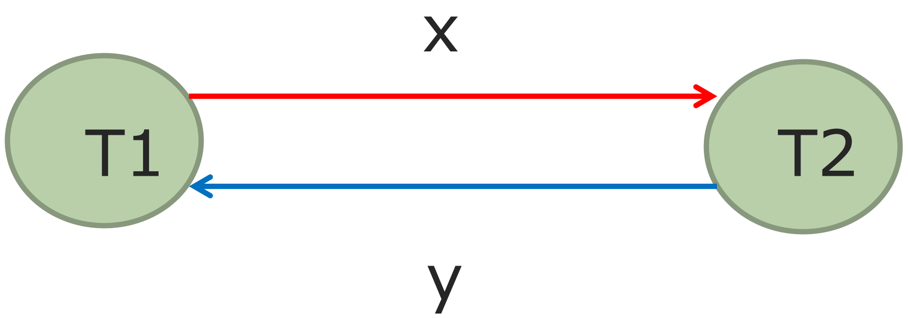


- Since there is no cycle, S is conflict serializable.

Example: Non-Conflict Serializable

- A schedule is **non-conflict serializable** if it is not conflict-equivalent to any other serial schedule.
- **Example: Non-serial schedule**

(T1, R(x)), (T1, W(x)), (T2, R(x)), (T2, W(x)), (T2, R(y)), (T2, W(y)), (T1, R(y)), (T1, W(y))



A directed edge (Ti → Tj) if Ti **writes** a value before Tj **reads/writes** it

A directed edge (Ti → Tj) if Ti **reads** a value before Tj **writes** it

- Precedence graph has a cycle, so the given schedule is non-conflict serializable schedule.

Example of Conflict Serializable Schedule

Time	T_7	T_8
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	write(bal_x)	
t_4		begin_transaction
t_5		read(bal_x)
t_6		write(bal_x)
t_7	read(bal_y)	
t_8	write(bal_y)	
t_9	commit	
t_{10}		read(bal_y)
t_{11}		write(bal_y)
t_{12}		commit

(a)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
	begin_transaction
	read(bal_x)
read(bal_y)	
	write(bal_x)
write(bal_y)	
commit	
	read(bal_y)
	write(bal_y)
	commit

(b)

T_7	T_8
begin_transaction	
read(bal_x)	
write(bal_x)	
read(bal_y)	
write(bal_y)	
commit	
	begin_transaction
	read(bal_x)
	write(bal_x)
	read(bal_y)
	write(bal_y)
	commit

(c)

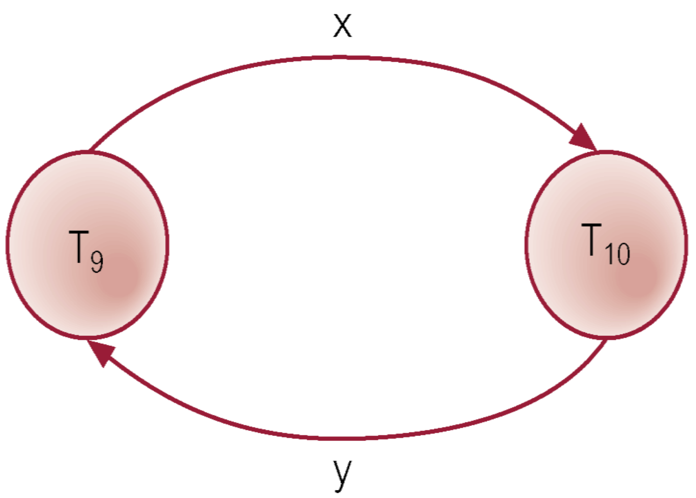
Equivalent schedules: (a) nonserial schedule S1

(b) nonserial schedule S2 equivalent to S1

(c) serial schedule S3, equivalent to S1 and S2

Example: Non-conflict Serializable Schedule

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x * 1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y * 1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	



A directed edge (Ti → Tj) if Ti **writes** a value before Tj **reads/writes** it

A directed edge (Ti → Tj) if Ti **reads** a value before Tj **writes** it

Main Point

If a schedule allows for the concurrent execution of two or more transactions, that schedule is serializable if the outcome is the same as it would be if the transactions were run serially (one after the other). In other words, the sequential behavior of a set of transactions defines the legal concurrent behaviors of transactions. Science & Technology of Consciousness: In nature the underlying sequential unfoldment of the unified field is at the basis of massive parallelism that is observed in nature.

Recoverability

- Serializability identifies schedules that maintain the database consistency, assuming that none of the transactions in the schedule fails.
- An alternative perspective examines the recoverability of transactions within a schedule if failure occurs.
- If transaction fails, *Atomicity* property requires that the effects of the transaction must be undone.
- *Durability* states that once transaction commits, its changes cannot be undone (without running another, compensating transaction).

Recoverable Schedule?

- Assume that instead of commit at the end, transaction T_9 fails and needs to be rolled back. Then what should happen?
- Should undo T_{10} because it has used a value for bal_x that has been rolled back!
- However, the Durability property does not allow this!
- So, this schedule is nonrecoverable schedule which shouldn't be allowed.

Time	T_9	T_{10}
t_1	begin_transaction	
t_2	read(bal_x)	
t_3	$bal_x = bal_x + 100$	
t_4	write(bal_x)	begin_transaction
t_5		read(bal_x)
t_6		$bal_x = bal_x * 1.1$
t_7		write(bal_x)
t_8		read(bal_y)
t_9		$bal_y = bal_y * 1.1$
t_{10}		write(bal_y)
t_{11}	read(bal_y)	commit
t_{12}	$bal_y = bal_y - 100$	
t_{13}	write(bal_y)	
t_{14}	commit	

Recoverable Schedule contd..

- A schedule where, for each pair of transactions T_i and T_j , if T_j reads a data item previously written (updated) by T_i , then the commit operation of T_i precedes the commit operation of T_j .
- With recoverability, we want to **make sure** that a **transaction** that **commits** has **not used dirty data!**
- To **express recoverability**, we **introduce** another **action** into the **schedule**:

  c_i = the transaction **T_i** ***commits***

Serializability & Recoverability

- ***Serializable* schedule**:
  - A schedule is *serializable* if the **effect** of the **execution** of the **actions** in the **schedule** is ***equivalent*** to a ***serial* schedule.**
- **Recoverable schedule:**
  - A schedule is recoverable if ***each* transaction commits** only ***after*** all **transactions** from which it has ***read*** have **committed**.
- ***Serializable* schedule** may **NOT** be ***recoverable*** !!!

In Class Exercise

- **Find out whether the following schedules are serializable and recoverable.**

1) S1 : T1(W,x), T1(W,y), T2(W,x), T2(R,y), c(T1), c(T2)

2) S2 : T2(W,x), T1(W,y), T1(W,x), T2(R,y), c(T1), c(T2)

3) S3 : T1(W,x), T1(W,y), T2(W,x), T2(R,y), c(T2), c(T1)

4) S3 : T2(W,x), T2(R,y), T1(W,x), T1(W,y), c(T2), c(T1)

UNITY CHART

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE:

Concurrency Control Protocols for Transaction Management

1. If a database is used concurrently by multiple users then care must be taken to assure that the users do not interfere with one another.
2. Concurrency control is necessary to ensure the isolation property of transactions.

---

3. **Transcendental consciousness** is the field of maximum correlation.
4. **Impulses within the Transcendental Field**: These impulses are perfectly balanced to create only the desired effect, no more and no less.
5. **Wholeness moving within itself:** In unity consciousness harmony predominates because everything is seen to be an expression of the Self.