



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David



TRANSACTIONS MANAGEMENT

31-Oct-2023

Week 5: Database Systems (DBS)
Dr Suresh Shanmugasundaram

So far...



1. ER Modeling
2. ER Models to Relational Tables
3. Functional Dependencies
4. Normalisation

Learning Objectives



1. Transactions & its properties
2. Concurrency & its problems
3. Schedules – serial and non-serial

Recap - Quiz



<https://forms.office.com/e/4ZmGkBjJ7n>

Database Transactions



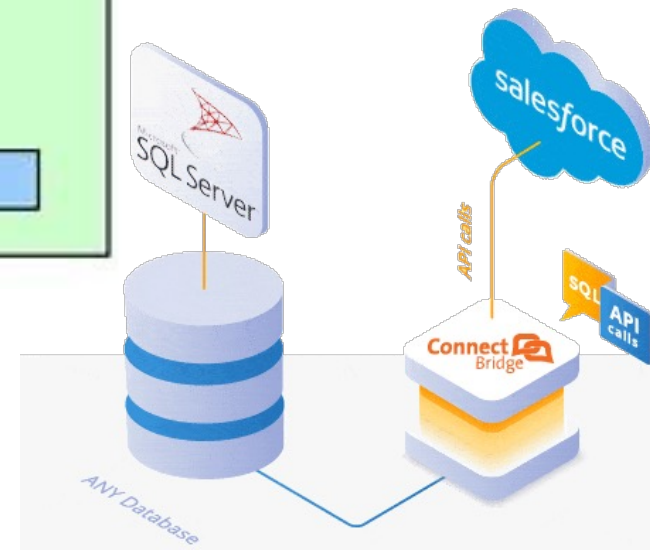
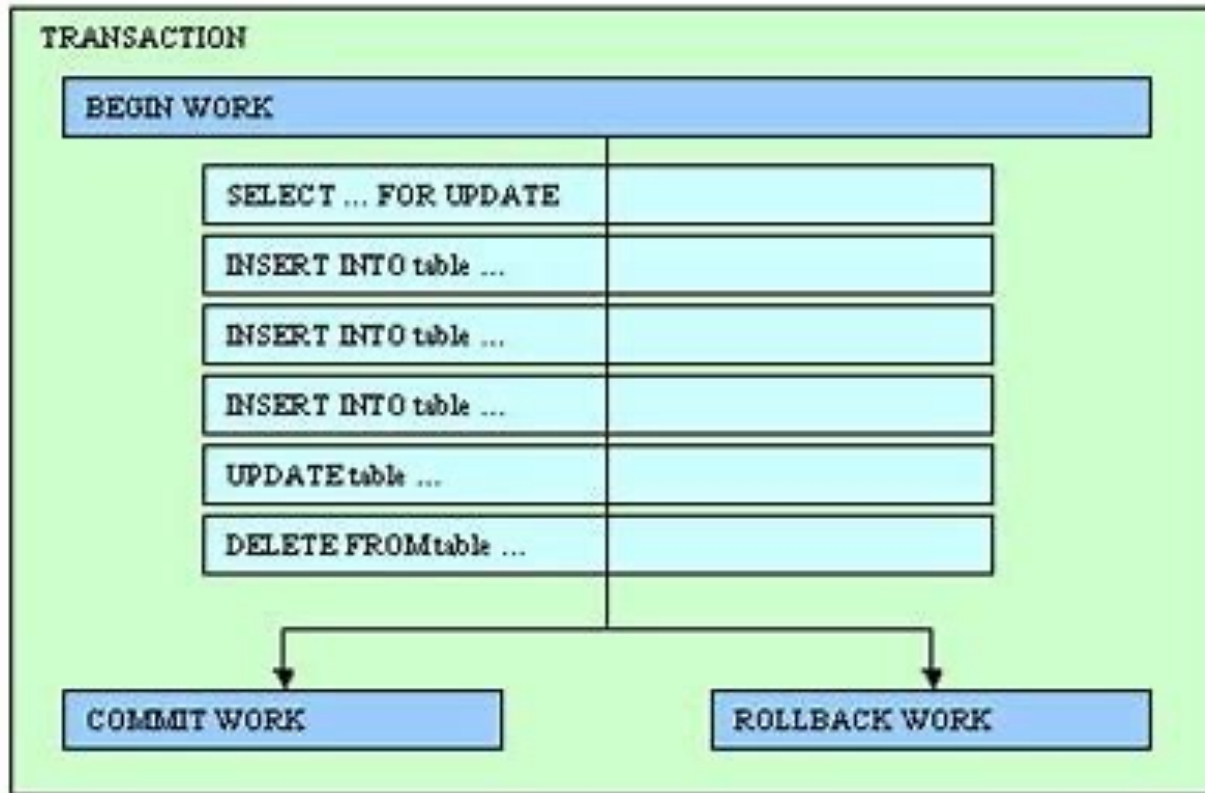
All parts of a transaction must be successfully completed to prevent _____? _____ problems.

Transactions

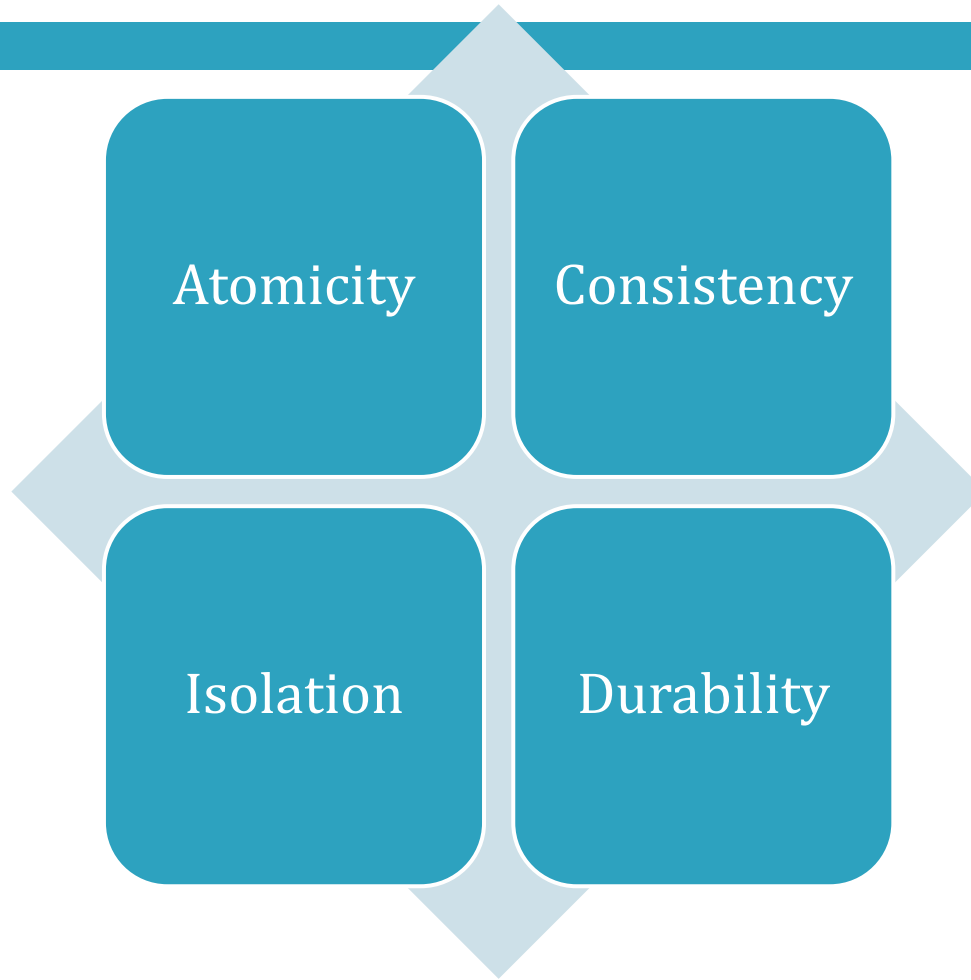
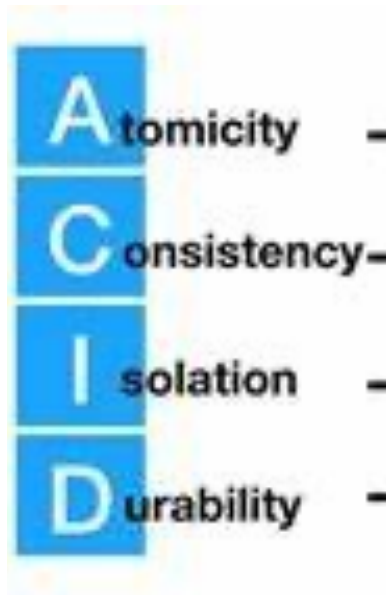
- Defined as any **one execution of a user program** in a DBMS
- Transaction is a unit of data processing or a list of actions.
 - E.g. At a bank → withdrawal or deposit of money, transfer of money from A's a/c to B's a/c.
- A Transaction would involve manipulation of one or more values in a database.



Transactions



Transaction Properties



Atomicity & Durability

□ Atomicity

- Defines a transaction to be a single unit of processing.

□ Durability

- If a transaction is committed, its changes to the database are ***permanent***, even if there is a system failure.

The DBMS component that ensures atomicity and durability is called the **recovery manager**.

Consistency & Isolation

Consistency

- Ensures that when the transaction is completed, the DB moves from one consistent state to another.

□ Isolation

- Concurrent transactions are ***not aware*** of each other. Each thinks that it is the only running transaction.
- In other words, the updates of a transaction should not be visible till they are committed.

Consistency & Isolation – Questions?

- Who is responsible for ensuring consistency?

One of the database components/users?

How/why?

- How is isolation guaranteed among several transactions?



Database Consistency

Transfer £50 from account A to B :

read(A)

$A = A - 50$

write(A)

read(B)

$B = B + 50$

write(B)

Example: fund transfer

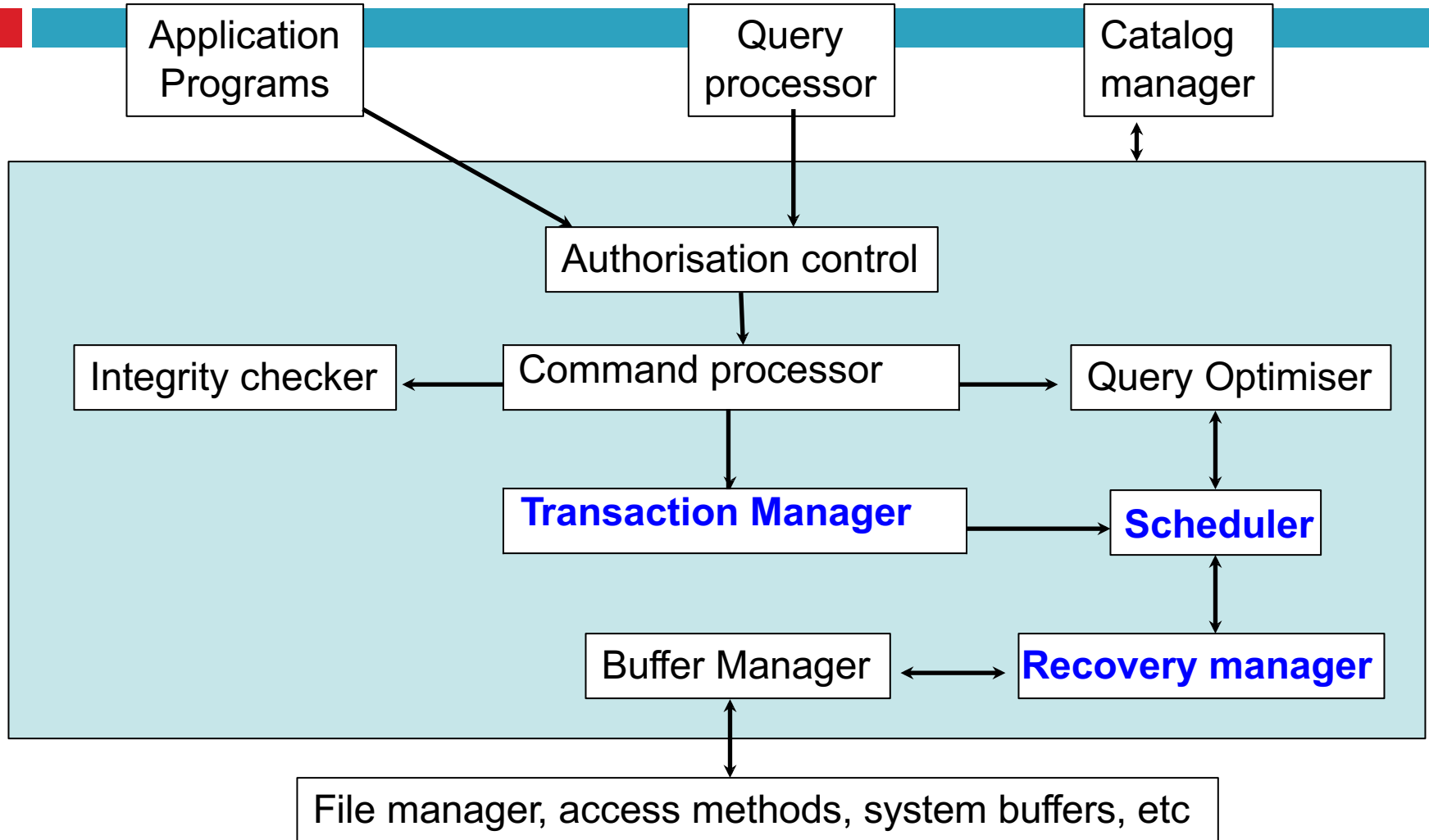
Transfer £50
from account *A*
to *B*:

```
read(A)  
A = A - 50  
write(A)  
read(B)  
B = B + 50  
write(B)
```

One transaction

- **Atomicity** – Shouldn't take money from *A* without giving it to *B*.
- **Consistency** – Money isn't lost or gained overall.
- **Isolation** – Other queries shouldn't see *A* or *B* change until completion.
- **Durability** – The money does not return to *A*, even after a system crash.

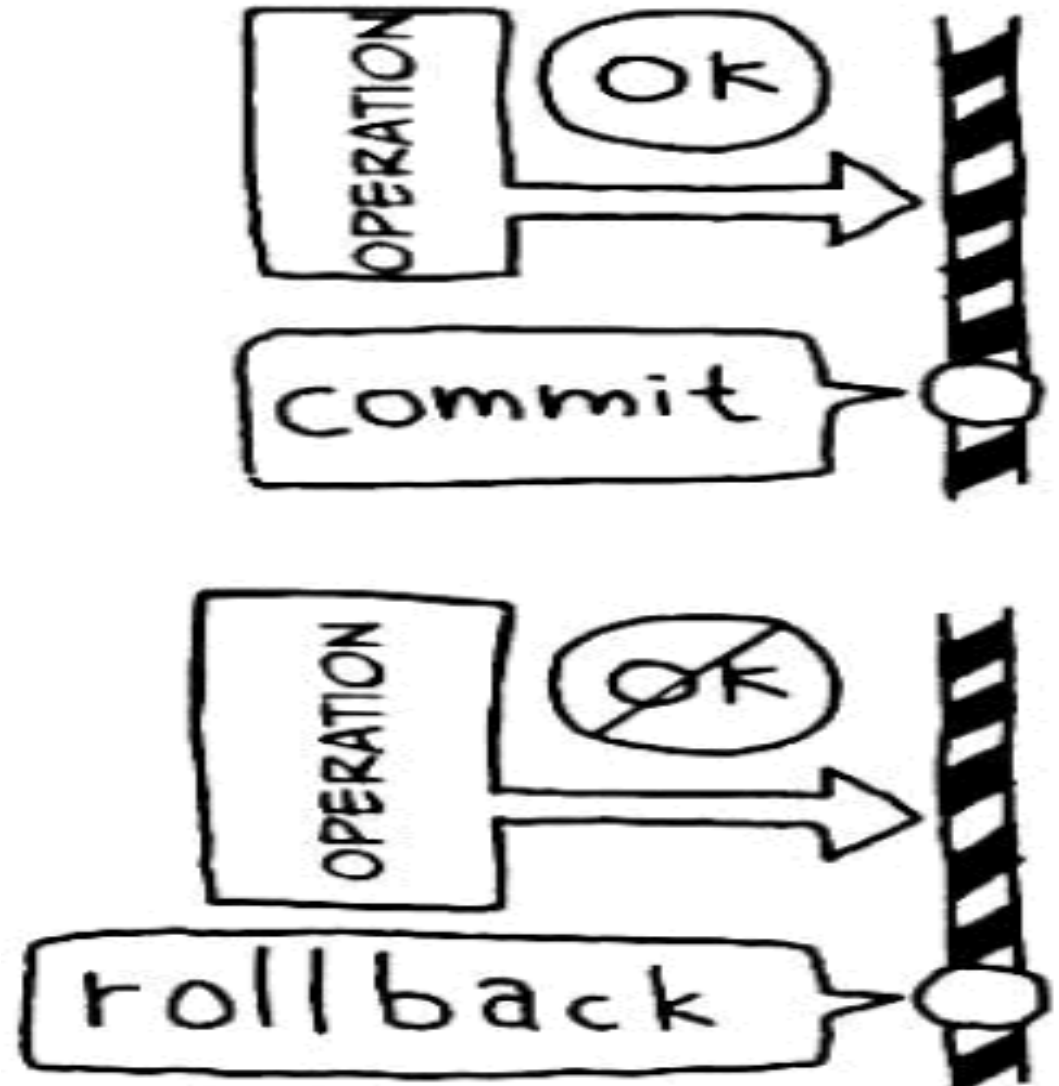
Database Manager



Transaction Manager

- The transaction manager enforces the ACID properties.
 - Schedules the operations of all transactions.
 - Uses COMMIT and ROLLBACK to ensure *atomicity*.
 - Locks or timestamps are used to ensure *consistency* and *isolation*.
 - A log is kept to ensure *durability*.

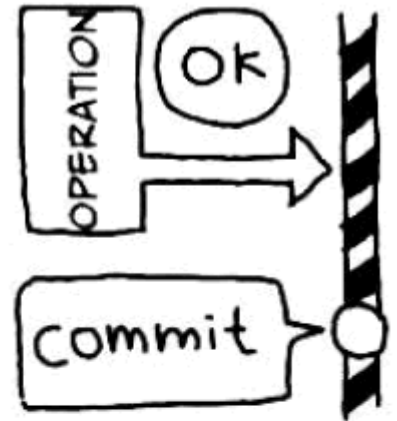
Atomicity



Atomicity

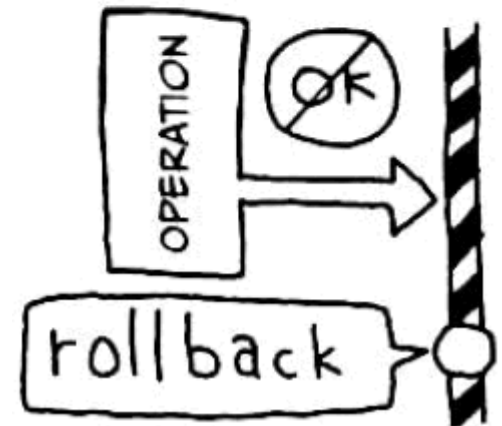
□ **COMMIT** - signal the *successful* end of a transaction

- Any changes that have been made to the database should be made permanent.
- These changes are now available to other transactions.



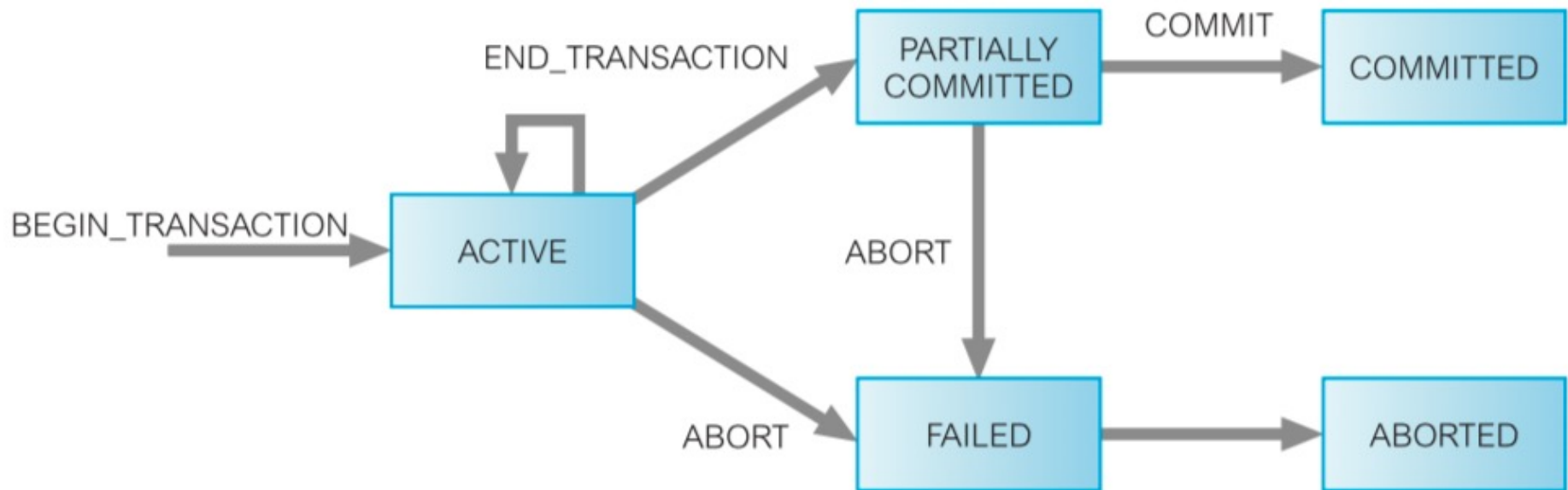
□ **ROLLBACK** - signal the *unsuccessful* end of a transaction

- Any changes that have been made to the database should be undone.
- It is now as if the transaction never happened, it can now be reattempted if necessary.



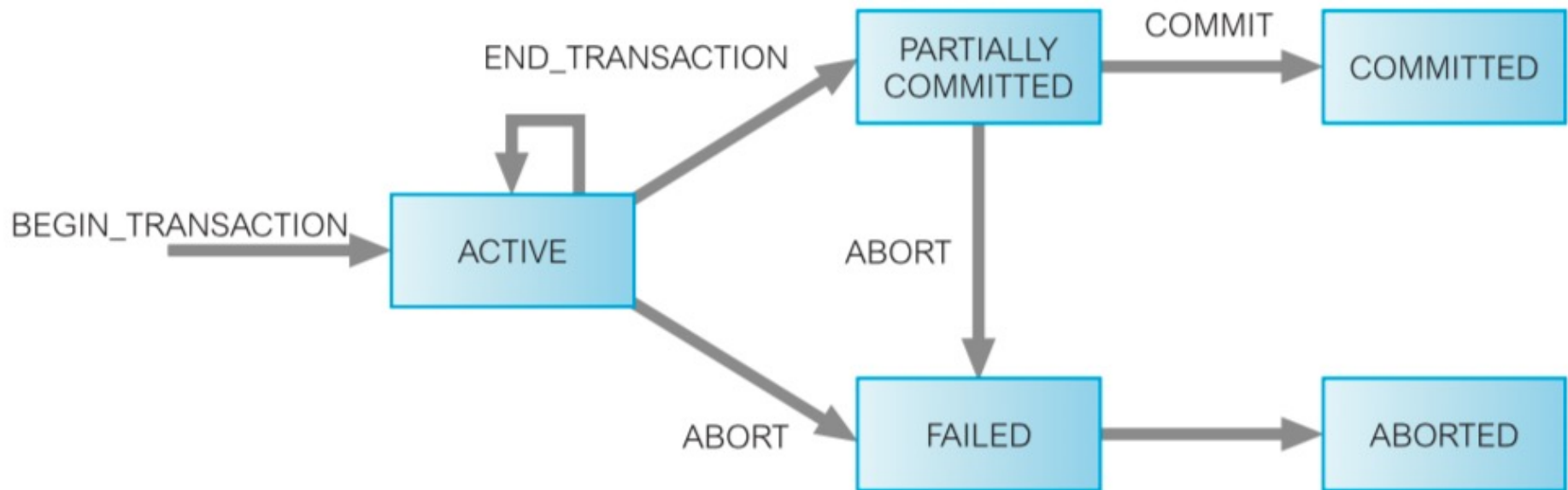
States of Execution

- ❑ A Transaction has many states of execution.
- ❑ State transition diagram for a transaction is



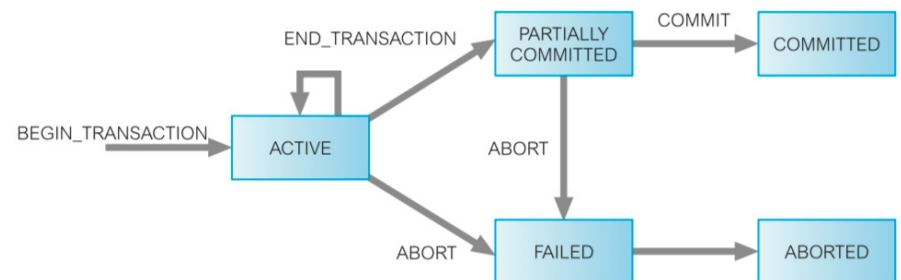
States of Execution

- ❑ A Transaction has many states of execution.
- ❑ State transition diagram for a transaction is



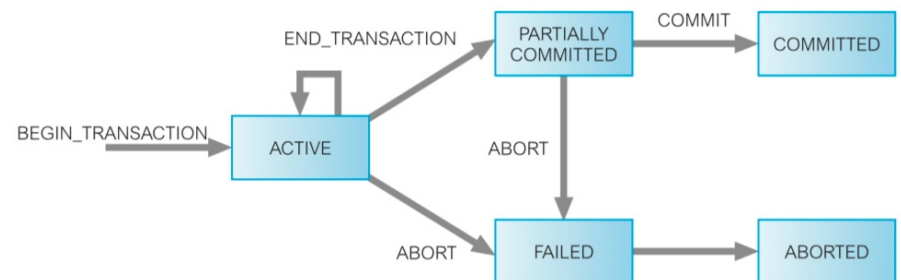
States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION	ACTIVE	1000	1000
READ(A);	PARTIALLY COMMITTED	?	?
A := A - 500;			
WRITE(A);			
READ(B);			
B := B + 500;			
WRITE(B);			
COMMIT;			
END TRANSACTION;			



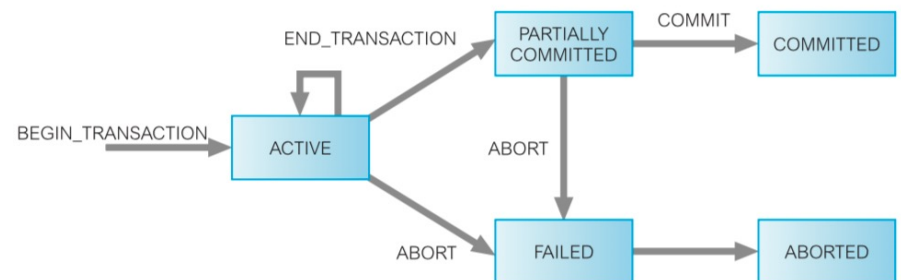
States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);			
READ(B);			
B := B + 500;			
WRITE(B);			
COMMIT;			
END TRANSACTION;			



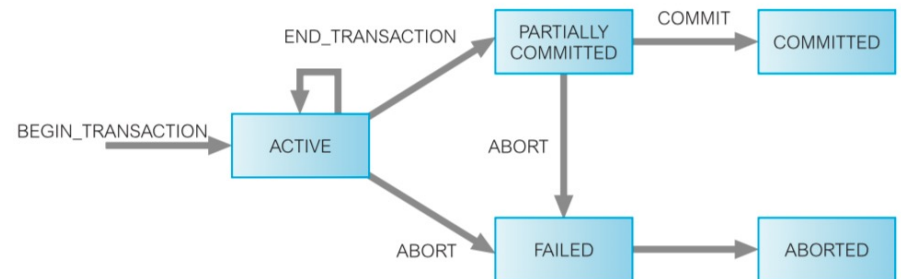
States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	?	?
READ(B);			
B := B + 500;			
WRITE(B);			
COMMIT;			
END TRANSACTION;			



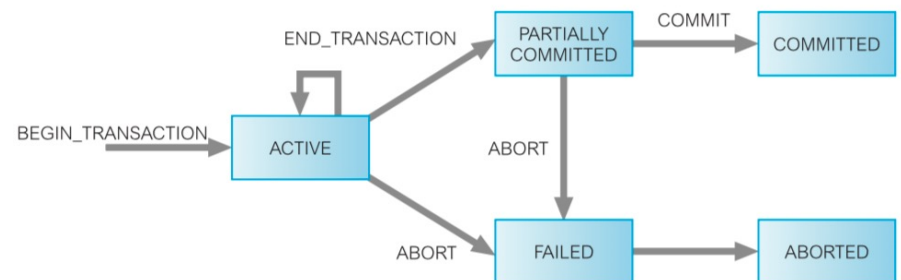
States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	?	?
WRITE(B);			
COMMIT;			
END TRANSACTION;			



States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;			
END TRANSACTION;			



States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;	FAILED before COMMIT and after WRITE(B)	?	?
END TRANSACTION;			

States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;	FAILED before COMMIT and after WRITE(B)	500	1500
END TRANSACTION;	FAILED state from PARTIALLY COMMITTED state	?	?

States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;	FAILED before COMMIT and after WRITE(B)	500	1500
END TRANSACTION;	FAILED state from PARTIALLY COMMITTED state	500	1500

States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;	FAILED before COMMIT and after WRITE(B)	500	1500
END TRANSACTION;			
	ABORTED state	?	?

States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;	FAILED before COMMIT and after WRITE(B)	500	1500
END TRANSACTION;	ABORTED state	1000	1000

States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;	FAILED before COMMIT and after WRITE(B)	500	1500
END TRANSACTION;	COMMITTED state	?	?

States of Execution - Example

Transaction T1	State	A	B
BEGIN TRANSACTION			
READ(A);	ACTIVE	1000	1000
A := A - 500;	PARTIALLY COMMITTED	500	1500
WRITE(A);	FAILED [before WRITE(A)]	1000	1000
READ(B);			
B := B + 500;	FAILED [after WRITE(A)]	500	1000
WRITE(B);			
COMMIT;	FAILED before COMMIT and after WRITE(B)	500	1500
END TRANSACTION;	COMMITTED state	500	1500

Transaction States and DML (MySQL)

- ❑ The following are the MySQL's DML (Data Manipulation Language) related to the Transaction Management.
- ❑ **BEGIN TRANSACTION:** *indicates the start of a new transaction.*
- ❑ **COMMIT:** *commits the current transaction, making the changes permanent.*
- ❑ **ROLLBACK:** *rolls back the current transaction, cancelling the changes made.*
- ❑ **SET autocommit={0|1}:** *disables/enables the default autocommit mode for the current session. Default is set to 1 (hence, it is set to autocommit)*

Concurrency

- Large databases are used by many people e.g. Banks!
 - Many transactions are to be run simultaneously
 - Faster than serial execution
 - Still need to preserve ACID properties
- Without concurrency
 - Easy to preserve atomicity and isolation
 - But have a long queue of transactions
 - Long transactions will delay others
 - e.g. do you want to queue at a counter on a busy day?

Concurrency Control

- ❑ **What?** The process of managing simultaneous operations on the database
- ❑ **Why?** when two or more users are accessing the database simultaneously and at least one is updating data, there may be interference that can result in inconsistencies.
- ❑ **How?** The operations of the two transactions are **interleaved**

Concurrency Problems

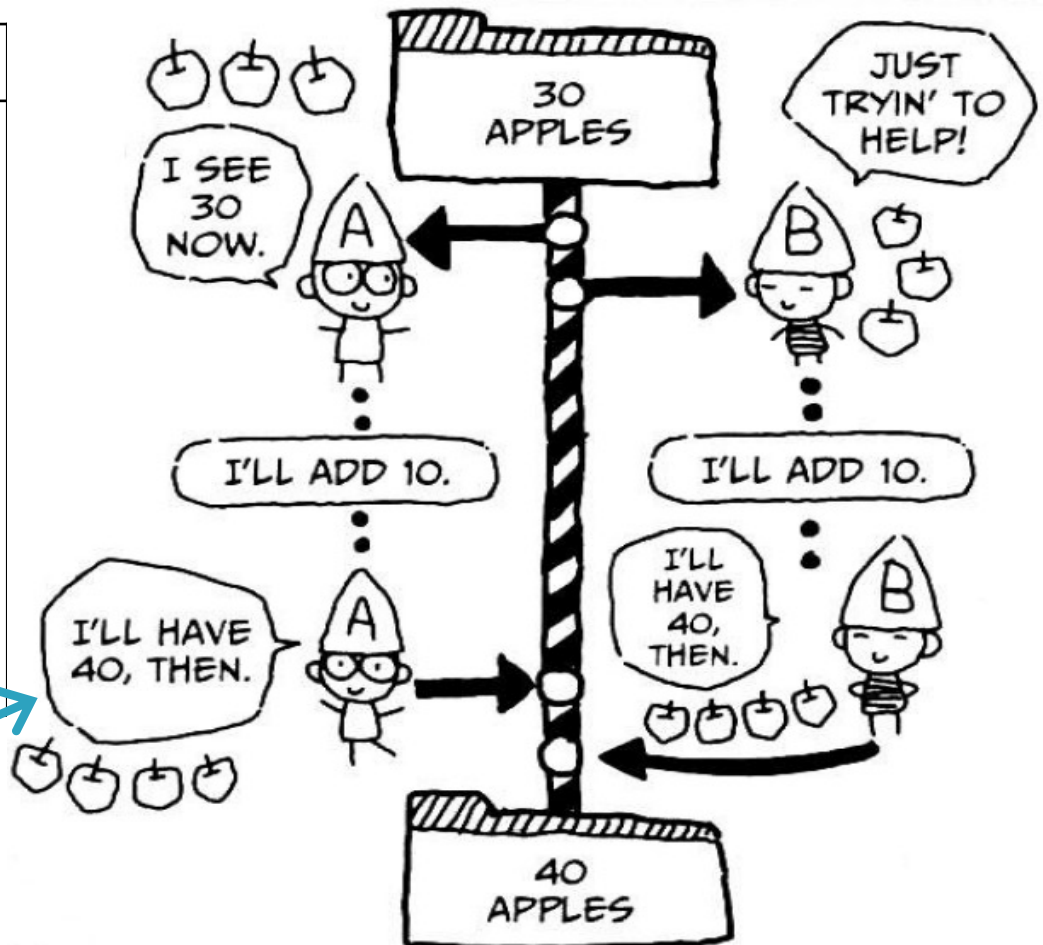
- ❑ The interleaving of operations in this way may produce an incorrect result, thus compromising the integrity and consistency of the database.
- ❑ The three examples of potential problems caused by concurrency:
 - a. **Lost update problem**
 - b. **Uncommitted dependency problem (aka Dirty Read problem)**
 - c. **Inconsistent Analysis problem**
- ❑ All arise when isolation is broken.

Lost Update Problem - Commit

- Successfully completed update operation by one user can be overridden by another user.

A (T1)	B (T2)
Read(X) $X = X + 10$	
	Read(X) $X = X + 10$
Write(X)	Write(X)
	COMMIT
COMMIT	

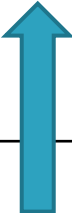
This update is lost!



there should be 50 APPLES afterwards!

Lost Update Problem - Rollback

T1	T2
Read(X) $X = X + 10$	Read(X) $X = X + 10$
Write(X)	ROLLBACK
COMMIT	



Let us assume that the initial value of X is 30

The **Rollback** causes the value of X to go back to the original value i.e. 30 instead of 40 for T1. When it is committing, it is committing to value 30 which was the original value.

Dirty Read Problem

- ❑ Occurs when one transaction can see the intermediate results of another transaction before it has committed.

T1	T2
Read(X)	
$X = X + 10$	
Write(X)	
	Read(X)
	$X = X + 10$
	Write(X)
ROLLBACK	
	COMMIT

A read that T2 should not have seen. It read a value that would never exist because of the rollback

Inconsistent Analysis Problem

- Occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.

T1 (X=5)	T2 (Y=10)
Read(X)	Read(Y) Y=Y+5 Write(Y) Read(X) X = X - 5 Write(X)
Read(Y) Sum=X+Y COMMIT	COMMIT

Summing up data while it is being updated. Happens when data items are used for analysis while another transaction is modifying it.

Other concurrency problems

- **Fuzzy Read:** Another problem can occur when a transaction T rereads a data item it has previously read but, in between, another transaction has modified it. Thus, T receives two different values for the same data item. This is sometimes referred to as a **nonrepeatable** (or **fuzzy**) **read**.
- **Phantom Read:** A similar problem can occur if transaction T executes a query that retrieves a set of tuples from a relation satisfying a certain predicate, re-executes the query at a later time but finds that the retrieved set contains an additional (**phantom**) tuple that has been inserted by another transaction in the meantime. This is sometimes referred to as a **phantom read**.

Concurrency Control

- The objective of a concurrency control protocol is to schedule transactions in such a way as to avoid any interference between them, and hence prevent the types of problem that might arise.
- One obvious solution is to **allow only one transaction to execute at a time**: one transaction is *committed* before the next transaction can *begin*.
- But that reduces the degree of concurrency in a multi-user DBMS.
- What then? Find a way to ensure the transactions that access different parts of the database can be scheduled together without interference. → Serialising the transactions.

Schedule(s)

- ❑ **Schedule:** A sequence of the operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.
- ❑ **Serial Schedule:** A schedule where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.
- ❑ **Non-serial Schedule:** A schedule where the operations from a set of concurrent transactions are interleaved.

Serial Schedule

Schedule 1:

$A = \text{£}1000$, $B = \text{£}2000$, $A + B = \text{£}3000$

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$ $\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$ commit

$A = \text{£}855$, $B = \text{£}2145$: $A + B = \text{£}3000$

Schedule 2:

$A = \text{£}1000$, $B = \text{£}2000$, $A + B = \text{£}3000$

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$ $\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$ commit

$A = \text{£}850$, $B = \text{£}2150$, $A + B = \text{£}3000$

Serial Schedule

Schedule 1:

$A = £1000, B = £2000, A + B = £3000$

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$ commit

$A = £855, B = £2145, A + B = £3000$

Schedule 2:

$A = £1000, B = £2000, A + B = £3000$

T_1	T_2
$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$\text{read}(A)$ $A := A - 50$ $\text{write}(A)$ $\text{read}(B)$ $B := B + \text{temp}$ $\text{write}(B)$ commit

$A = £850, B = £2150, A + B = £3000$

Both transactions lead to the data being in a consistent state but **NO** CONCURRENT

Non-Serial Schedule

Schedule 3:

A=£1000, B=£2000: A+B=£3000

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

A=£855, B=£2145: A+B=£3000

Schedule 3 is not a serial one, but it is **equivalent** to the serial schedule 1.

Ensures concurrency and leaves the system in a consistent state.

Schedules – Not always lucky

A=£1000, B=£2000: A+B=£3000

T_1	T_2
$\text{read}(A)$ $A := A - 50$	$\text{read}(A)$ $\text{temp} := A * 0.1$ $A := A - \text{temp}$ $\text{write}(A)$ $\text{read}(B)$
$\text{write}(A)$ $\text{read}(B)$ $B := B + 50$ $\text{write}(B)$ commit	$B := B + \text{temp}$ $\text{write}(B)$ commit

Not equivalent to any serial
schedule – **PROHIBITED!**

The job of the database management system is to ensure that the state is consistent.

A=£950, B=£2100: A+B=£3050

Next up...



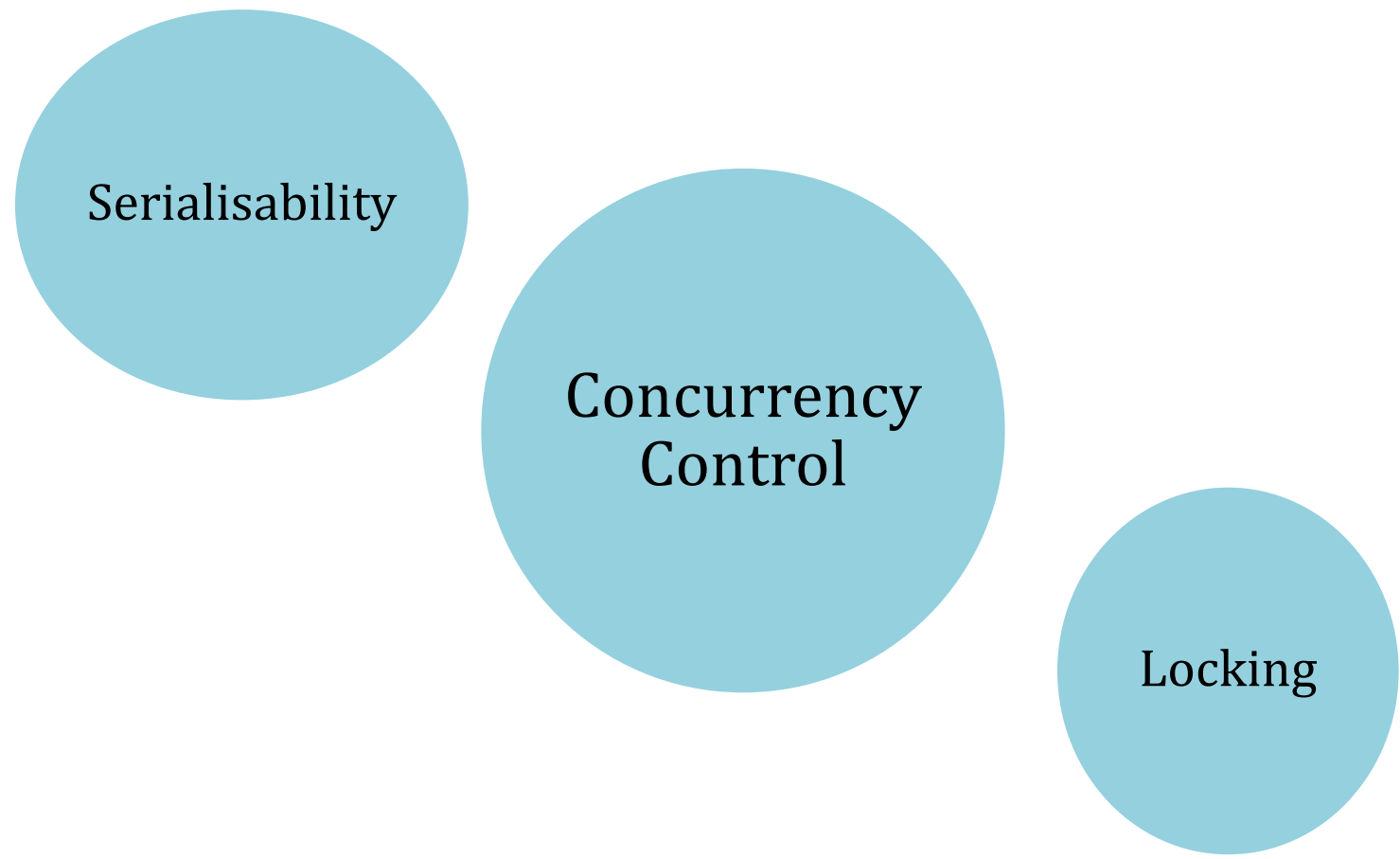
- How to non-serialise schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.

Summary



- Transactions
- Properties of Transactions
- States of execution of a transaction
- Concurrency Control
- Problems associated with concurrency control

Coming up...



Serialisability

Concurrency
Control

Locking

Lab Exercise



- ❑ Download Instructions from Moodle

Coming up...



ACID Properties