Transactions and Concurrency

In production DBs users perform transactions concurrently.

- DBMS wants to maximize throughput
- Without compromising integrity

Example:

Person tries to remove, at the same time $100 from bank account.

T_1	T_2
Read balance	Read balance
If balance > 100	If balance > 100
subtract 100 from account	subtract 100 from account

time ↓

Can we have reach a state where person gets $200 but bank only records $100 given?

If so, we have lost consistency of data.

Properties of Transactions:

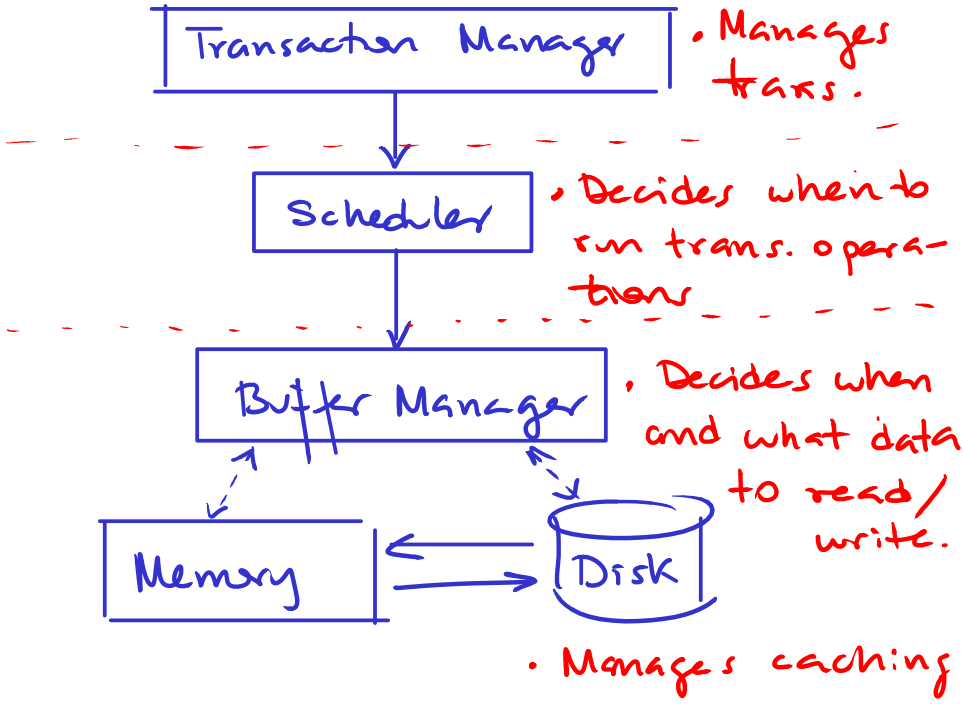
ACID

- **Atomicity**: A transaction happens in its entirety or not all all
  Incomplete transactions must be undone.
- **Isolation**: A transaction must appear to be executed as if no other transaction is executing at the same time.
  Transactions cannot communicate with each other.
- **Durability**: The effect on the db of a transaction has successfully completed must never be lost.
  - Even in the event of failures.

Responsability of Programmer.

- **Consistency**: Transactions are given a DB in a consistent state and are expected to keep it consistent.

The role of the DBMS is to maximize number of concurrent trans. while maintaing ACID.



To maximize throughput, the scheduler might:

- delay transaction operations.
- reorder transactions

To guarantee ACID, the scheduler:

- must make sure transactions are durable
- avoid undesirable interleaving of trans.
- deal with deadlocks

Transactions

Any transaction either
- completes (commits)
  or
- aborts (rollback)

Atomicity.

If system crashes: (server or client):
- Non completed transactions must be undone (rollback) Durability.

Correctness Principle

Any transaction, if executed in isolation will transform any consistent state of the DB into another consistent state.

The DBMS must guarantee isolation even when many trans. are executed concurrently

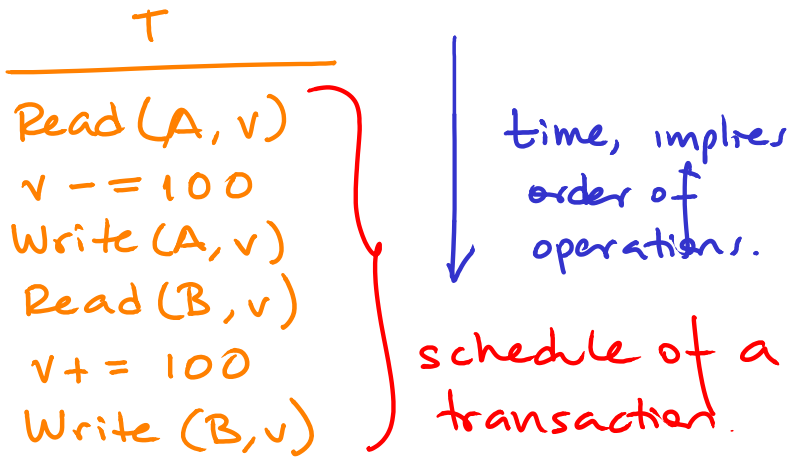
A transaction is a list of actions.
For simplicity sake we will only consider read/write of DB objects.

Notation:

Read (A, v) Reads DB object A into local variable v (local to transaction)

Write (A, v) Replaces DB object A with value in v.

Ex: T is a transaction that moves $100 from account A to account B:



There might be many copies of the same transaction running.

Ex: Two instances of T are trying to run simultaneously.

Assumption:

Reads and Writes are atomic and cannot be interleaved.

Schedule

Sequence of actions taken by one or more transactions.

When two transactions want to be executed 3 options:

1) T_1 executes first, then T_2

denoted:

2) $T_1 ; T_2$, $T_2 ; T_1$ } Serial schedules.

3) The operations of T_1 and T_2 interleave.

Many, many possible interleavings of operations of T_1 and T_2

- Some safe
- Some unsafe (break consistency)

Serial Schedule

A schedule is serial if its actions consists of all the actions of one trans. followed by all the actions of another transaction and so on.

Ex.

T_1	T_2
Read (A, t)	
t += 100	
Write (A, t)	
	Read (A, s)
	s *= 1.1
	Write (A, s)

$T_1 ; T_2$

T_1	T_2
	Read (A, s) s *= 1.1 Write (A, s)
Read (A, t) t += 100 Write(A, t)	

$T_2 ; T_1$

Each schedule might have a different impact on DB.

Say A_0 value of A before schedule.

$T_1 ; T_2 \Rightarrow A = 1.1 (A_0 + 100)$

$T_2 ; T_1 \Rightarrow A = 100 + 1.1 A_0$

Serializable Schedule.

A schedule S is serializable if there exists a serial schedule S' of the same transactions such that for every initial state of the DB, the effect of S and S' is the same.

T_1	T_2
Read (A, t)	
	Read (A, s)
t+ = 100	
write (A, t)	
	s * = 1.1
	Write (A, s)

Effect of schedule:

$A = 1.1\,A \neq$ effect of $T_1; T_2$ or $T_2; T_1$.

$\Rightarrow$ non-serializable.

Another schedule:

T_1	T_2
Read (A, t)	
t+.=100	
write (A, t)	
	Read (A, s)
	s*=1.1
	Write (A, s)

Serializable:
Equivalento to $T_1; T_2$

To model transactions we only care about Read, Write, Commit, Rollback.

We can rewrite the schedle above as:

$R_1(A), W_1(A), R_2(A), W_2(A), C_2, C_1$

Use A_i for rollback (abort).

The job of the DBMS is to only allow serializable schedules.

Anomalies due to interleaved execution

There are 3 main ways in which 2 interleaved transactions can leave the DB in an inconsistent state.

Two actions on the same data object conflict if at least one of them is a write.

1) Read Uncommitted Data

Dirty Read

- T_1 writes to object A
- Before T_1 ends T_2 reads A.

2) Unrepeatable Read

- T_1 reads an object
- T_2 changes the value of the same object before T_1 ends.

3) Blind Write (Overwriting not committed data)

- T_1 updates the value of A
- T_2 overwrites the value of A without before reading it, before T_1 ends.

$\Rightarrow$ Lost update.

Ex:

T_1 moves 100 from A to B

T_2 increases both A and B by 10%

Serial schedules:

$T_1 ; T_2$ $\quad A = 1.1\ (A_0 - 100)$

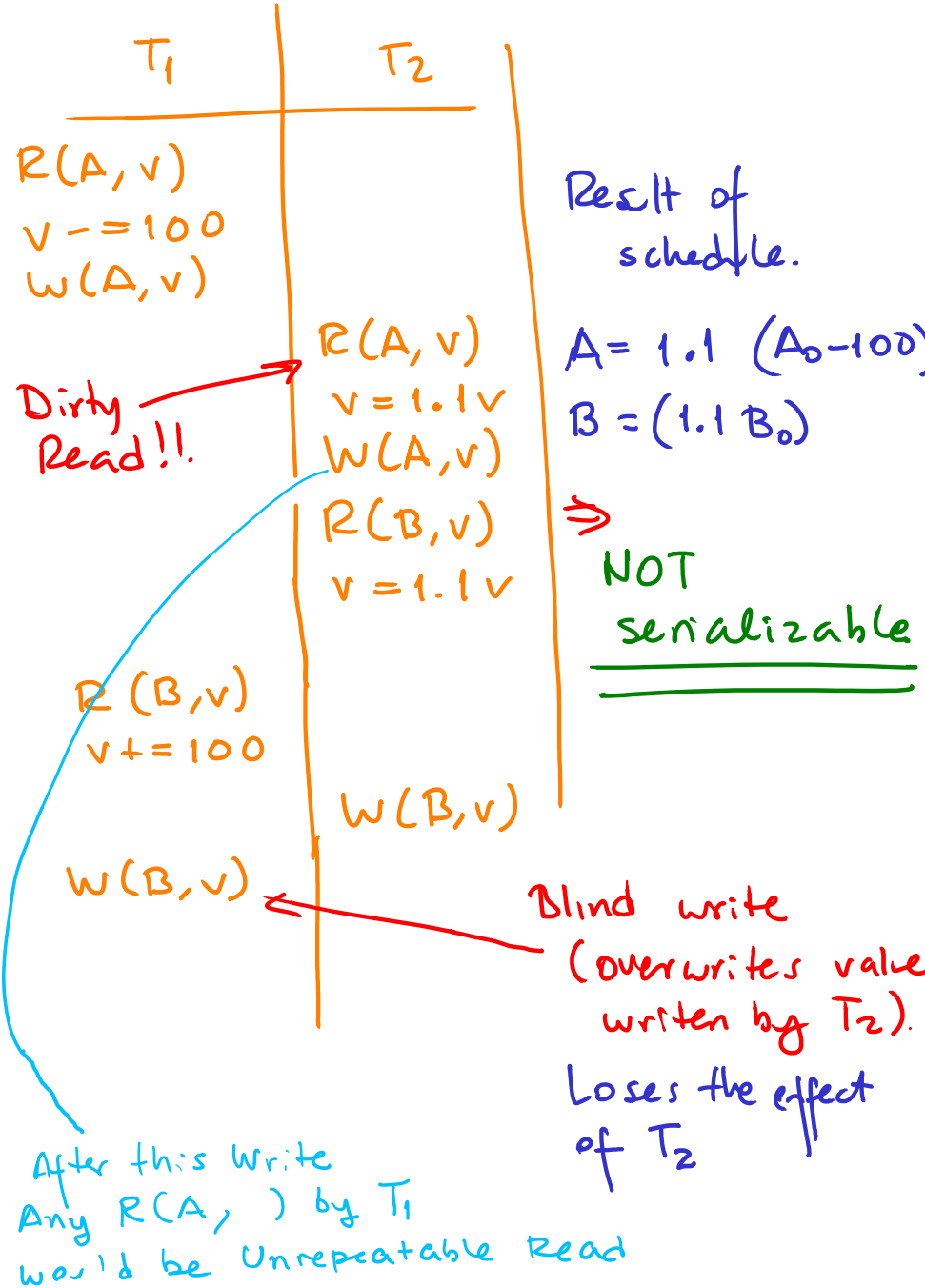
$B = 1.1\ (B_0 + 100)$

or

$T_2 ; T_1$ $\quad A = (1.1\ A_0) - 100$

$B = (1.1\ B_0) + 100$

Ex: A non serial schedule:



A transaction with anomalies might still result in a serializable schedule.

but...

the DBMS will not execute transactions with anomalies... unless indicated otherwise. (See next notes)