# ACID in details

# Overview of this video

The video will give a precise definition of each of the four ACID properties, i.e.

**A: Atomicity**

**C: Consistency**

**I: Isolation**

**D: Durability**

# A - Atomicity

A transaction is an **atomic unit of processing**

- An indivisible unit of execution
- Executed in its entirety or not at all

Deals with failure (“aborts”)

- User aborts transaction (e.g., cancel button)
- System aborts transaction (e.g., deadlock)
- Transaction aborts itself (e.g., unexpected database state)
- System crashes, network failure, etc.

# A - Atomicity

Abort - an error prevented full execution

- We **UNDO** the work done up to the error point
  - System re-creates the database state as it was before the start of the aborted transaction

Commit - no error, entire transaction executes

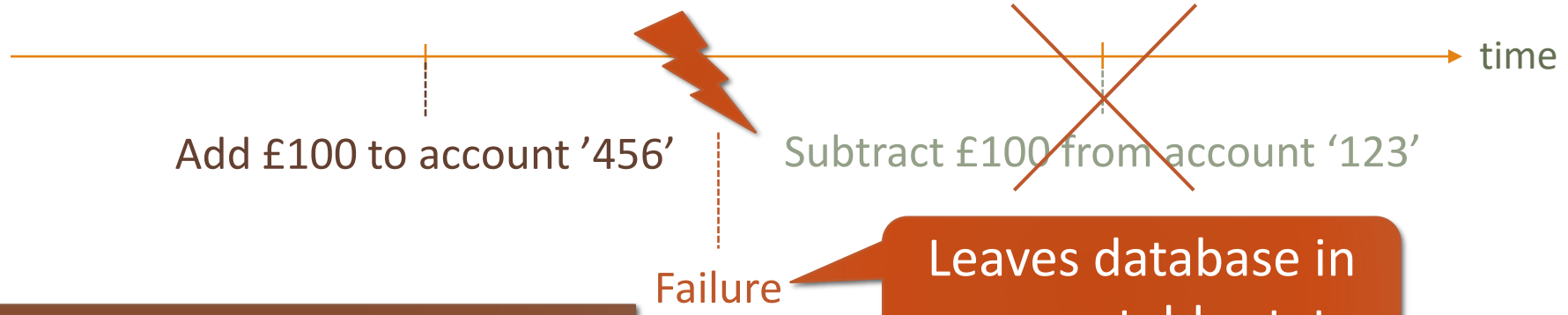
- The system is updated correctly

# Problem 2: Partial Execution

(from the introduction to transactions video)

**Accounts**(accountNo, accountHolder, balance)

Goal: Transfer £100 from account '123' to account '456'

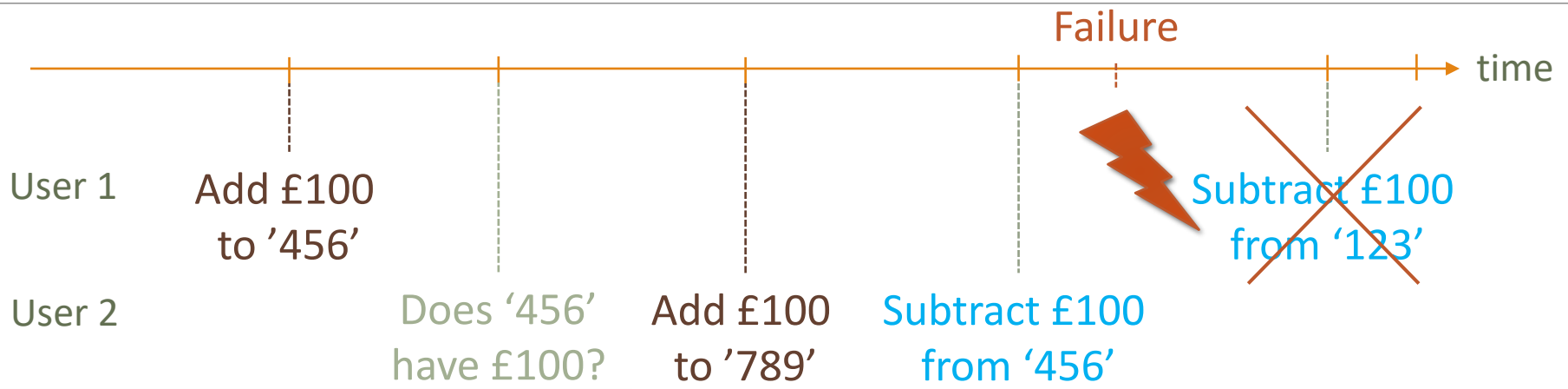



```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 456;
```

```
UPDATE Accounts
SET    balance = balance - 100
WHERE  acountNo = 123;
```

# Problem 3 (Concurrency & Partial Execution)

(from good schedules/transactions video)




```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 456;
```

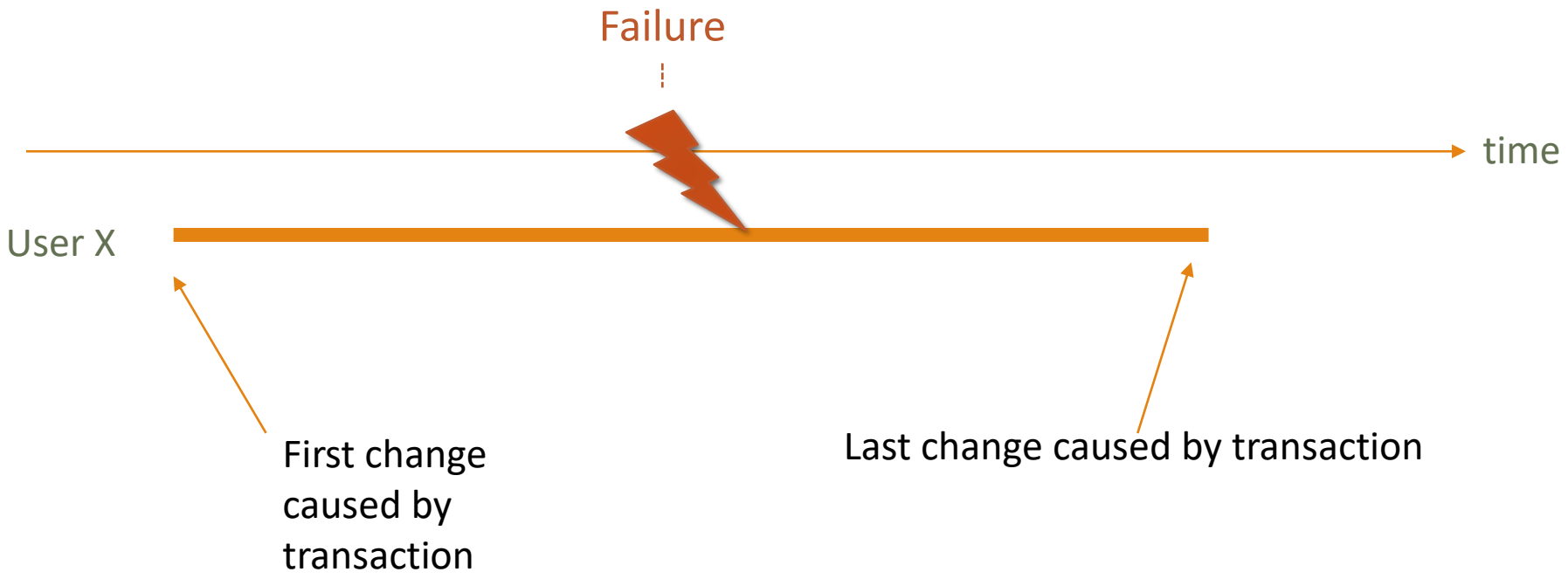
```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 456;
```

```
SELECT balance
FROM   Accounts
WHERE  accountNo = 456;
```

```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 789;
```

```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 123;
```

# Atomicity is broken



# C - Consistency

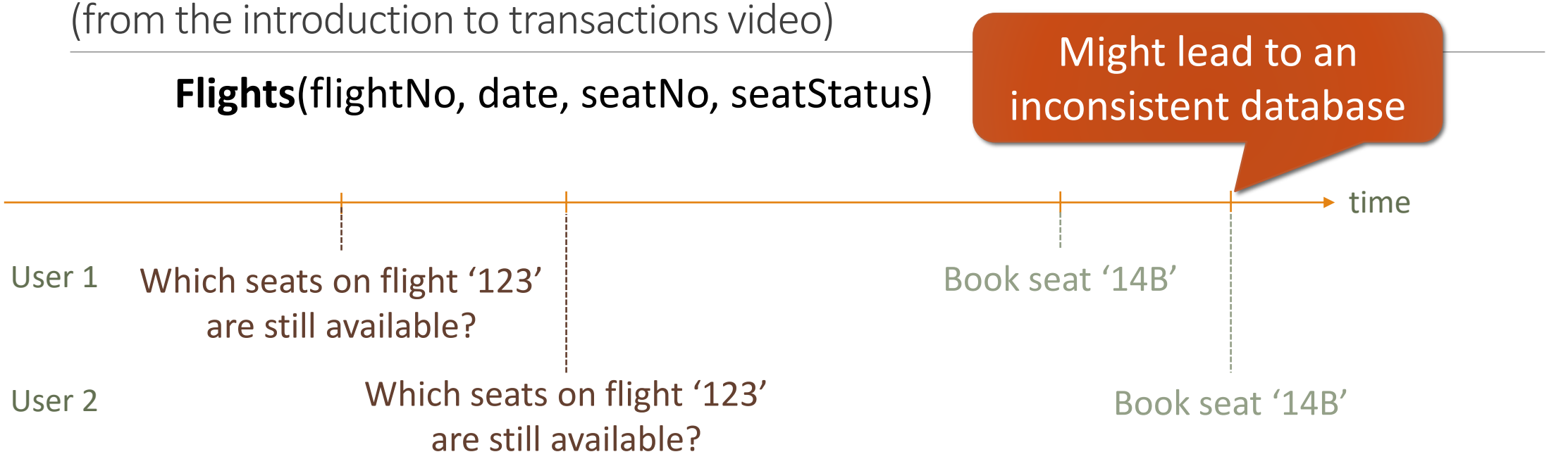
A correct execution of the transaction must take the database from one consistent state to another

- Weak version: Transactions may not violate constraints (typical definition)
- Strong version: It should correctly transform the database state to reflect the effect of a real world event (ISO/IEC 10026-1:1998)

# Problem 1: Concurrency

(from the introduction to transactions video)

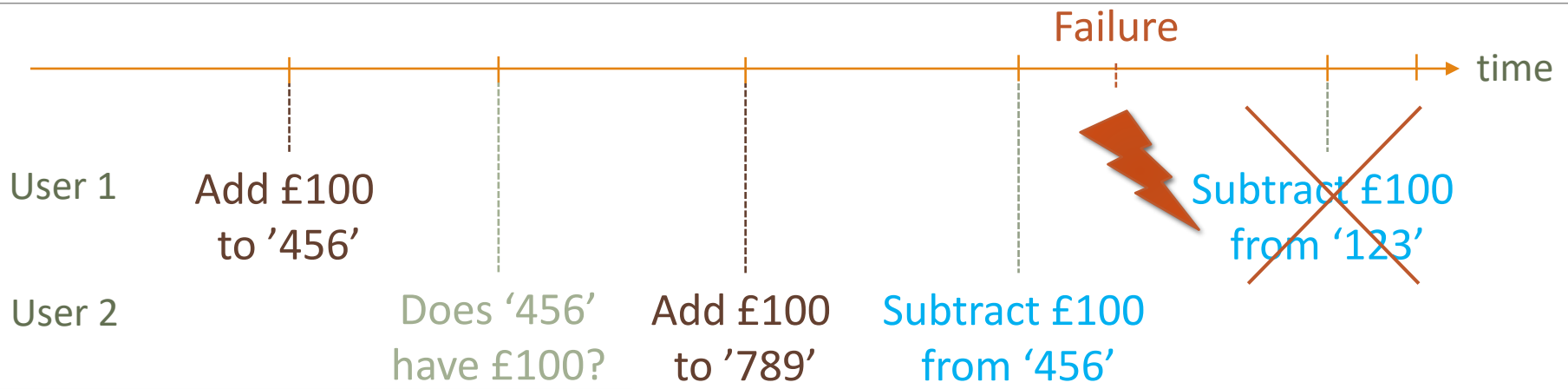
**Flights**(flightNo, date, seatNo, seatStatus)




```
SELECT seatNo
FROM   Flights
WHERE  flightNo = 123
   AND date = '2020-10-30'
   AND seatStatus = 'available';
```

```
UPDATE Flights
SET    seatStatus = 'occupied'
WHERE  flightNo = 123
   AND date = '2020-10-30'
   AND seatNo = '14B';
```

# Problem 3: Concurrency & Partial Execution




```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 456;
```

```
SELECT balance
FROM   Accounts
WHERE  accountNo = 456;
```

```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 789;
```

```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 456;
```

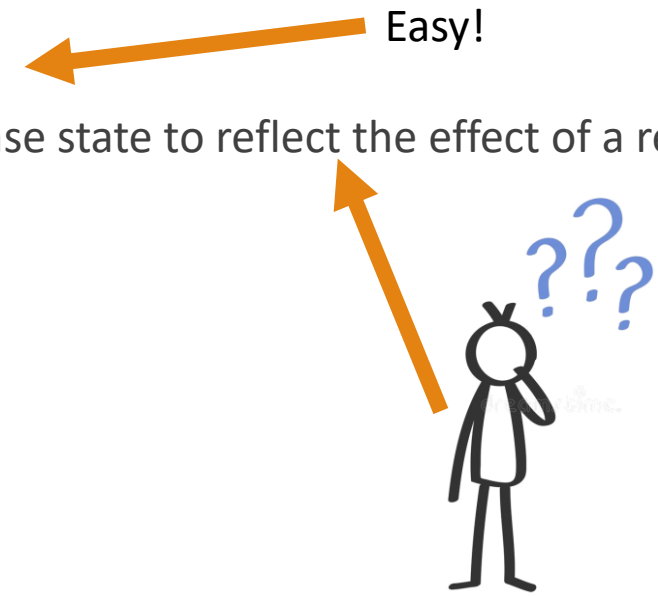
```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 123;
```

# Consistency is tricky

From definition:

A correct execution of the transaction must take the database from one consistent state to another

- Weak version: Transactions may not violate constraints
- Strong version: It should correctly transform the database state to reflect the effect of a real world event



# Transactions Preserve Consistency

Fundamental assumption:

> Transactions (if run on their own) always transform a *consistent* database state into another *consistent* database state.

They produce one of two outcomes

- Commit (i.e. Successful)
  - Execution was successful and database is left in a consistent state
- Abort (i.e. Failed)
  - Execution was not successful and we need to restore the database to the state it was in before execution

This implies that serial schedules are consistent!

- Or more generally, schedules that has changes the database in the same way as a serial schedule are Consistent – such schedules are called ***serializable***

# I - Isolation

A schedule satisfies isolation iff it is **serializable** (I.e. like last slide: the effect of a serializable schedule is the same as some serial schedule)

- Note: consistency is not really defined by serializable, but in this course, we will consider serializable to imply consistency
- On the other hand, a schedule is serializable iff it satisfies isolation

# Weakening isolation

The SQL standard divides Isolation into how strongly you want it:

1. Read uncommitted (basically, not having Isolation at all): It is fine if you read data which has not been committed
2. Read committed: Everything you read must have been committed before you can see it
3. Repeatable read: … and If you read the same thing twice in a transaction, you must get the same return value
4. Serializable (formally, they also want that the earlier levels are satisfied, so it is a bit stronger)

Some implementations can have more levels!

Isolation level 4 is the default meaning of Isolation

You can decide:

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```

Can also be: READ ONLY

Alternately: READ COMMITTED, REPEATABLE READ, SERIALIZABLE

# Problem 1: Concurrency

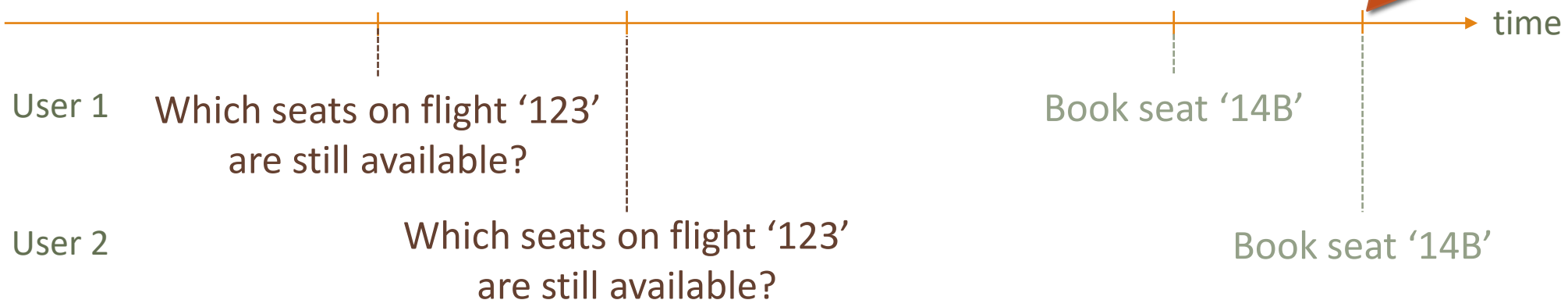
(from the introduction to transactions video)

REPEATABLE READ depends on implementation

This schedule does satisfy READ COMMITTED, but is not SERIALISABLE

**Flights**(flightNo, date, seatNo, seatStatus)

Might lead to an inconsistent database

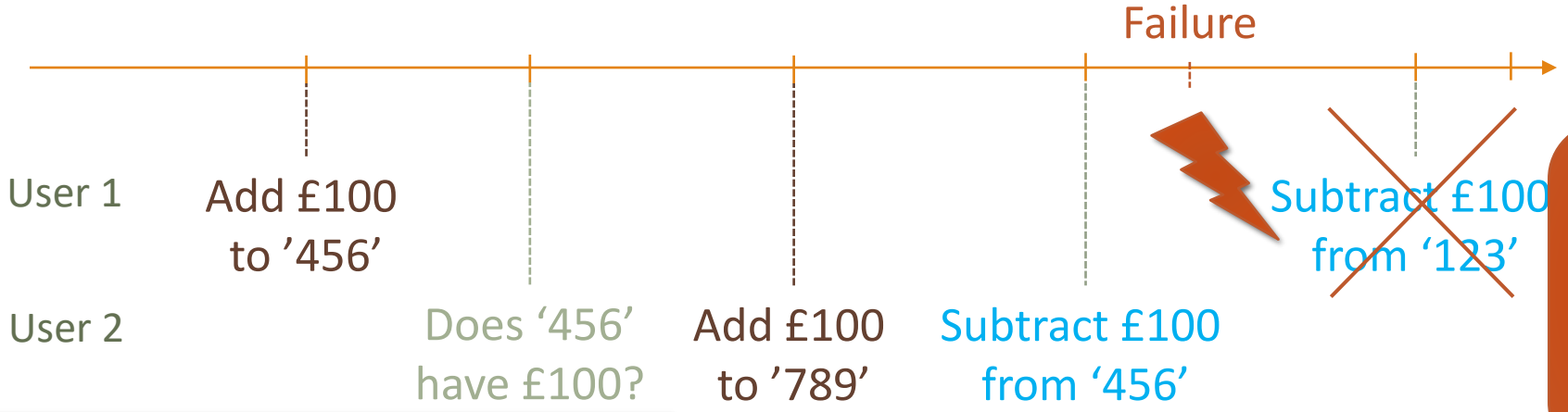



```
SELECT seatNo
FROM   Flights
WHERE  flightNo = 123
   AND date = '2020-10-30'
   AND seatStatus = 'available';
```

```
UPDATE Flights
SET    seatStatus = 'occupied'
WHERE  flightNo = 123
   AND date = '2020-10-30'
   AND seatNo = '14B';
```

# Problem 3 (Concurrency & Partial Execution)

(from good schedules/transactions video)




This schedule does satisfy READ UNCOMMITTED, but not READ COMMITTED

```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 456;
```

```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 456;
```

```
SELECT balance
FROM   Accounts
WHERE  accountNo = 456;
```

```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 789;
```

```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 123;
```

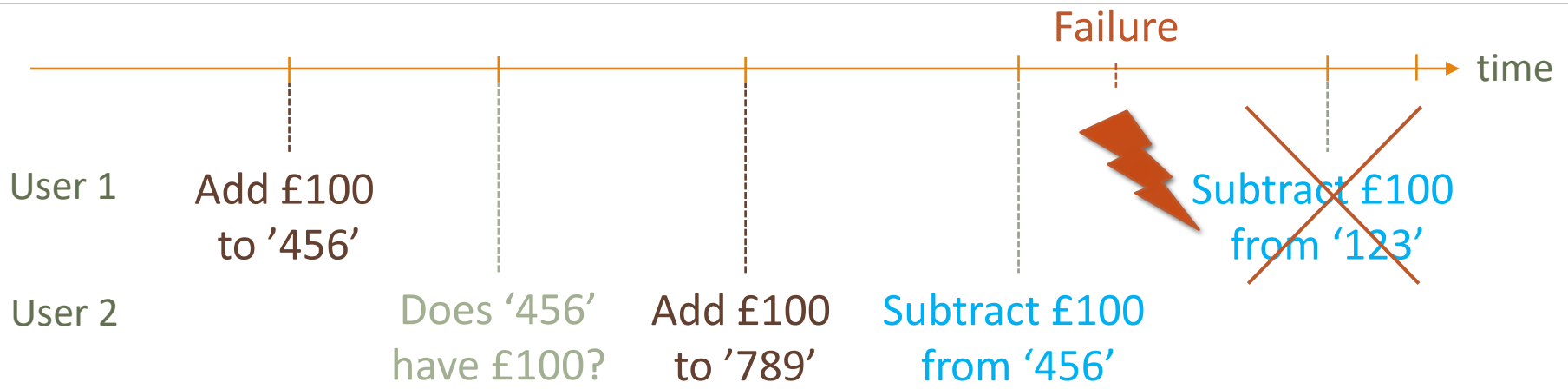
# D - Durability

Once a transaction commits and changes the database, these changes cannot be lost because of subsequent failure

- The effect of a transaction on the database should not be lost after the commit point
  - But could be overwritten by a later update
- We **REDO** the transaction if there are any problems after the update
- Durability deals with things like media failure

# Problem 3 (Concurrency & Partial Execution)

(from good schedules/transactions video)




```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 456;
```

```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 456;
```

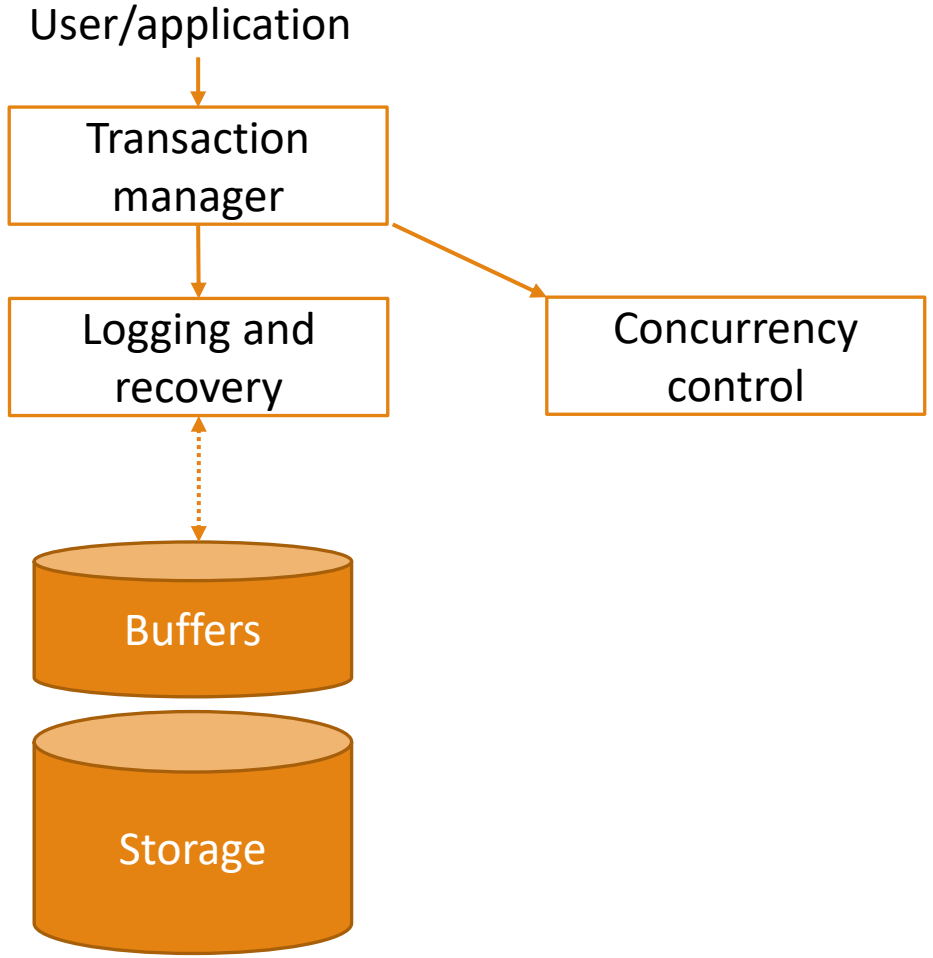
```
SELECT balance
FROM   Accounts
WHERE  accountNo = 456;
```

```
UPDATE Accounts
SET    balance = balance + 100
WHERE  accountNo = 789;
```

```
UPDATE Accounts
SET    balance = balance - 100
WHERE  accountNo = 123;
```

# Some relational DBMS Components

(part of a slide in the content video)



# Transaction - ACID Properties

We had two components in transaction management:

1. Concurrency control
2. Logging and recovery

What components is used to satisfy ACID

**A: Atomicity**
- via Recovery Control (Logging and Recovery)

**C: Consistency**
- via Scheduler – Concurrency Control

**I: Isolation**
- via Scheduler – Concurrency Control

**D: Durability** (or permanency)
- via Recovery Control

# Summary

DBMS's are expected to avoid certain issues

Specifically, they should not violate the 'ACID' Properties:

**A:** **Atomicity – Everything or nothing**

**C:** **Consistency – somewhere between constraints are satisfied, up to matches the real world**

**I:** **Isolation – The effect matches each transaction executing alone in some order**

**D:** **Durability – Transactions that have finished does not later get undone**