Being able to recover

Overview

In this video, we will go back and look at one of the initial three problems we discussed when defining ACID and how to deal with it

Conflict-Serialisability vs Recovery

CONFLICT-SERIALISABILITY

Many nice properties:

- Equivalent to serial schedules
- Ensure consistency / correctness

Can be enforced by two-phase locking (2PL)

LOGGING AND RECOVERY

Suitable logging techniques ensure that we can restore desired database states

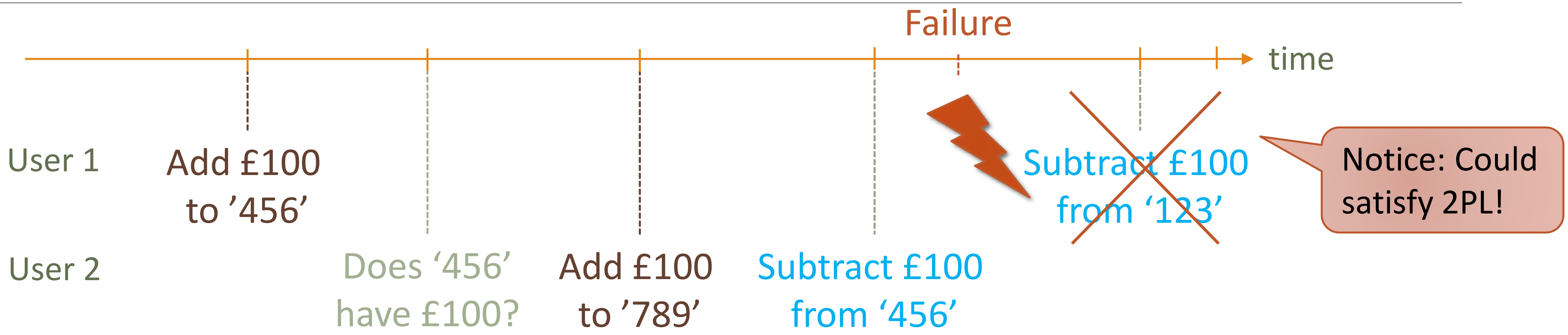
- Undo incomplete transactions
- Redo committed transactions
- Undo a single or a selected number of transactions

Robust: works even after system failures

Problem: cascading rollbacks may be necessary!

Problem 3: Currency & Partial Execution

(from the good schedules/transaction video)




```
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;
```

No good solutions to the problem

(adapted from the good schedules/transaction video)

Let us look at some options:

- We could do nothing, but then the bank lost money
  - This would break **Atomicity**
- We could undo the first transaction, but not the second, but then the second transaction might not be valid anymore, because there could be too little money on the account to transfer 100£
  - This would break **Consistency,** because we would break an integrity constraint
  - This would break **Isolation** as well (at least on some levels)
  - (it is also inconsistent – not the property - with what abort should do)
- We could undo both transactions, but the second one have finished and the person doing it might have already gone away (because everything looked good when he finished)
  - This would break **Durability**

“Dirty Reads”

In practice, the isolation property is often not fully enforced (→ “dirty reads” may occur)

Reason: efficiency!

- Spend less time on preventing ”dirty reads”
- Gain ”more parallelism” by executing some transactions that would have to wait to prevent “dirty reads”

Other option:
READ ONLY

Other levels in SQL:
READ COMMITTED,
REPEATABLE READ,
SERIALIZABLE

You can decide:

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

“Dirty reads” can slow down the system when transactions have to abort

Cascading Rollback

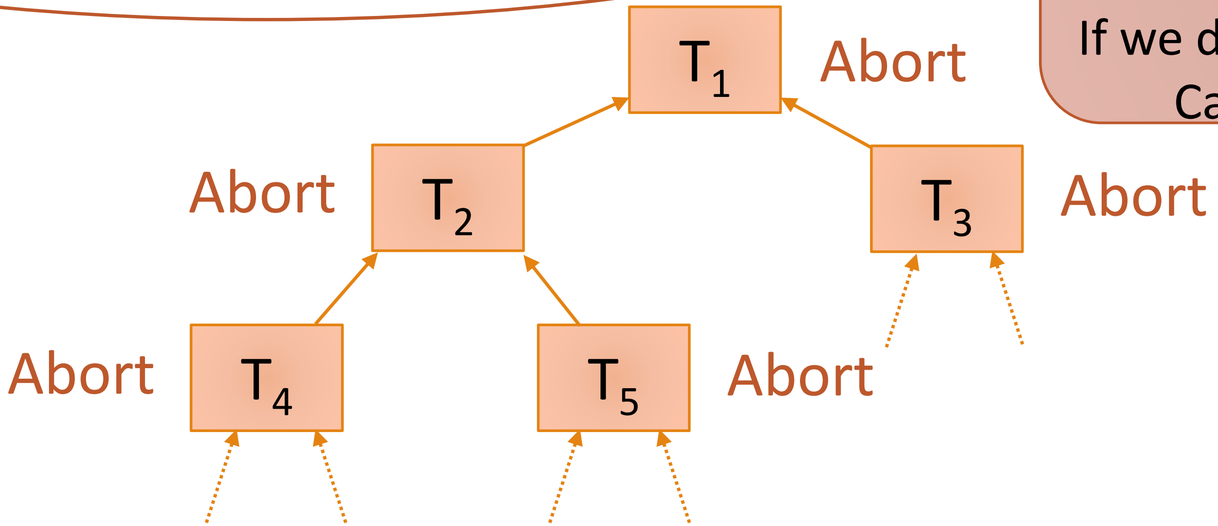
If a transaction T aborts:

- Find all transactions that have read items written by T.
- Recursively abort all transactions that have read items written by an aborted transaction.

Very slow → want to avoid this

If we do **not abort** all these:
- Break Isolation (Inconsistent with what abort should do)

If we do **abort** them:
- Can break Durability



Isolation vs Durability

Time	Transaction T_1	Transaction T_2	X	Y
0	lock(X)		1	2
1	read_item(X)			
2	X := X + 100		101	
3	write_item(X)			
4	lock(Y)			
5	unlock(X)			
6		lock(X)		
7		read_item(X)		
8		X := X * 2	202	
9		write_item(X)		
10		**commit**		
11	read_item(Y)			
12	**abort**			

If we do **not abort** T_2:
Break Isolation
(Inconsistent with what abort should do)
If we do **abort** T_2:
Can break Durability

Recoverable Schedules

Can still do cascading rollbacks, but only active transactions can be forced to abort

The problem for Durability in regards to cascading rollbacks occur because a transaction T_2 reads data from some transaction T_1, then T_2 commits and afterwards T_1 aborts.

A schedule S is **recoverable** if the following is true:

- if a transaction T_1 commits and has read an item X that was written before by a different transaction T_2, …
- then **T_2 must commit before T_1 commits**.

Example

A **recoverable** schedule:

$$S_1: w_2(X); w_1(Y); w_1(X); r_2(Y); w_2(Y); c_1; c_2$$

T_2 reads data that was written before by T_1

T_1 must commit before T_2 can commit

A **non-recoverable** schedule:

$$S_2: w_1(X); w_1(Y); w_2(X); r_2(Y); w_2(Y); c_2; c_1$$

T_2 reads data that was written before by T_1

But: T_2 commits first

Note:

- S_1 is *not* serialisable.
- S_2 is serialisable.

Recoverable Schedules – implicit assumption

Additional implicit requirement:
All log records have to reach disk in the order in which they are written.

Compare:

- Recoverable: S_1: $w_2(X); w_1(Y); w_1(X); r_2(Y); w_2(Y); c_1; c_2$
- Not recoverable: S_3: $w_2(X); w_1(Y); w_1(X); r_2(Y); w_2(Y); c_2; c_1$

If in S_1 the commit record for T_2 would reach disk earlier than the commit record for T_1, then T_1 could in principle abort → cascading rollback

Summary

Reconciliation of conflict-serialisability and recovery

- Can lead to problems (**cascading rollbacks**) if done naively
- Avoiding cascading rollbacks requires a smarter way of scheduling transactions

Ideas:

- **Recoverable schedules**: T commits only if all transactions that T has read from have committed