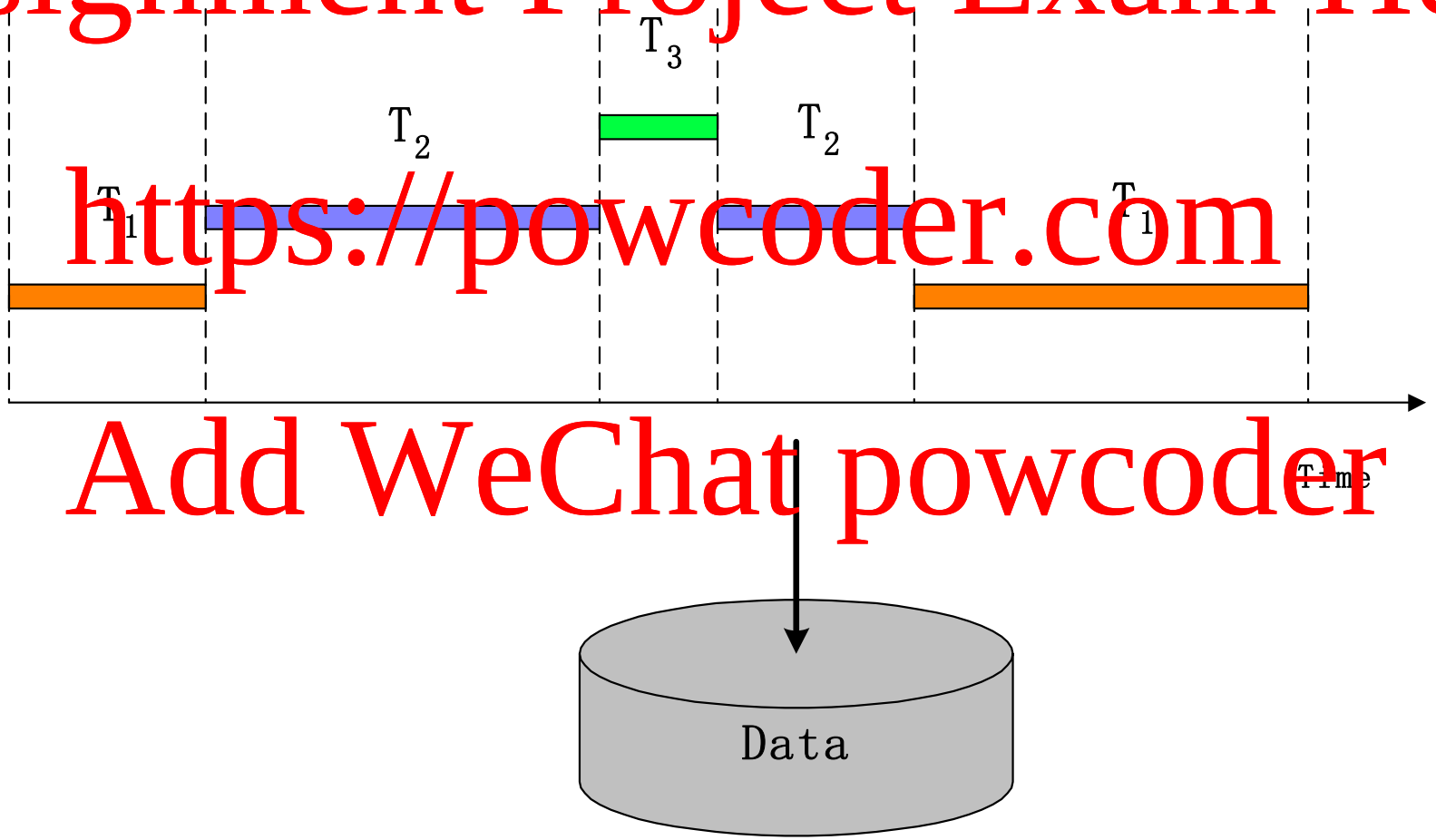
Database Transactions – Part 3

Concurrent Transactions

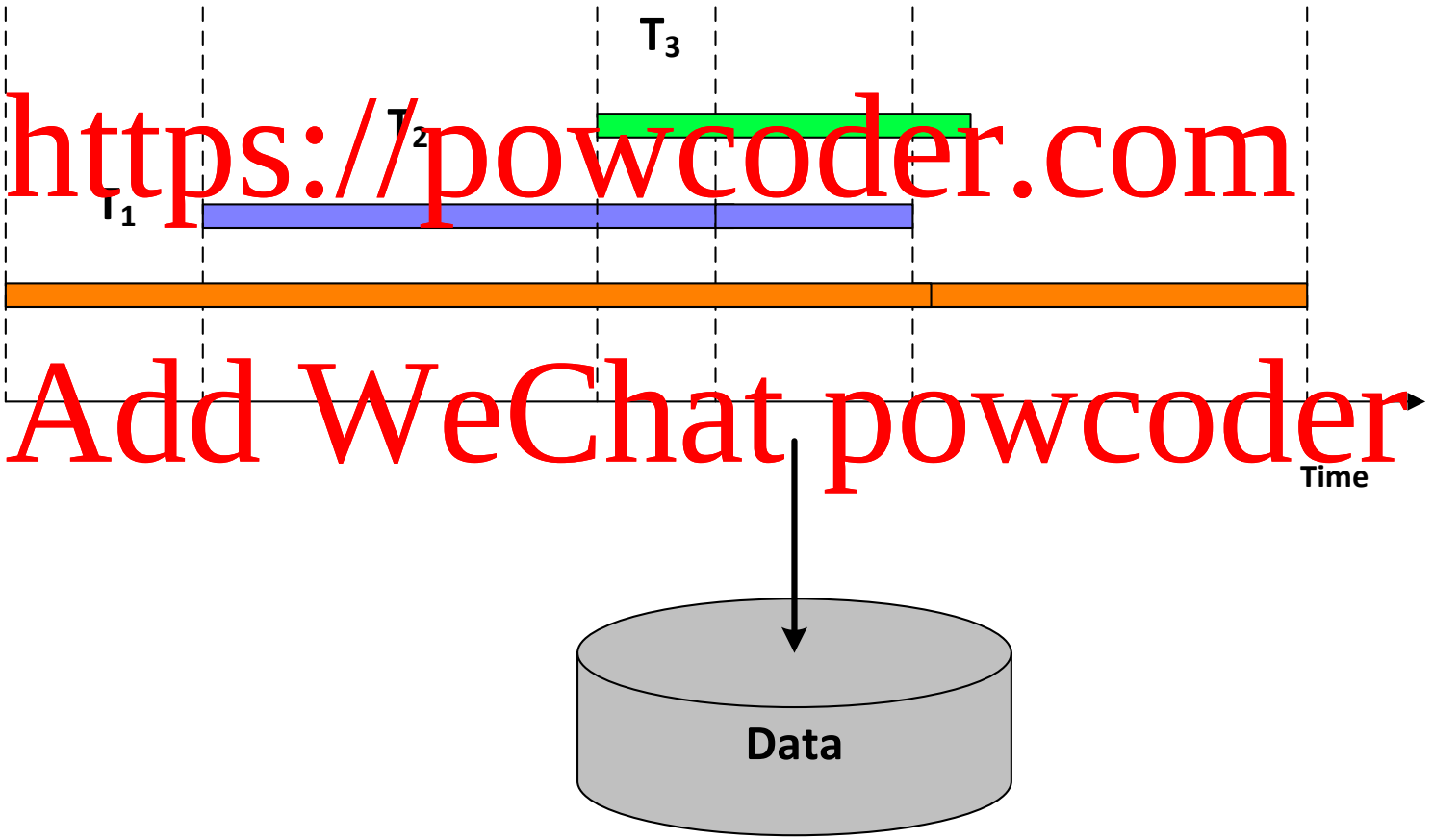
Concurrent Transactions

- **Interleaved processing**: transactions are interleaved in a single CPU.



Concurrent Transactions

- **Parallel processing**: transactions are executed in parallel in multiple CPUs.



Concurrent Transactions

- Executing transactions concurrently will improve database performance
  - ⇝ **Increase throughput** (*average number of completed transactions*)
    - For example, while one transaction is waiting for an object to be read from disk, the CPU can process another transaction (because I/O activity can be done in parallel with CPU activity).
  - ⇝ **Reduce latency** (*average time to complete a transaction*)
    - For example, interleave execution of a short transaction with a long transaction usually allows the short one to be completed more quickly.

- But the DBMS has to guarantee that the interleaving of transactions **does not lead to inconsistencies**, i.e., **concurrency control**.

Why is Concurrency Control Needed?

- Concurrency control is needed for preventing the following problems:

1. The **lost update** problem
2. The **dirty read** problem
3. The **unrepeated read** problem
4. The **phantom read** problem

(1) - The Lost Update Problem

- **Example:** Bob withdraws $100 from his account ($T_1$) while Alice deposits $500 into Bobs account ($T_2$).


```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T1: COMMIT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```


Steps	T_1	T_2
1	read(B)	
2		read(B)
3	write(B) (B:=B-100)	
4	commit	
5		write(B) (B:=B+500)
6		commit

Steps	B(Bob)
before 1	$200
after 2	$200
after 4	$100
after 6	$700

- **Question:** What is the problem?

(1) - The Lost Update Problem

- **Example:** Bob withdraws $100 from his account ($T_1$) while Alice deposits $500 into Bobs account ($T_2$).


```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T1: COMMIT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```


Steps	T_1	T_2
1	read(B)	
2		read(B)
3	write(B) (B:=B-100)	
4	commit	
5		write(B) (B:=B+500)
6		commit

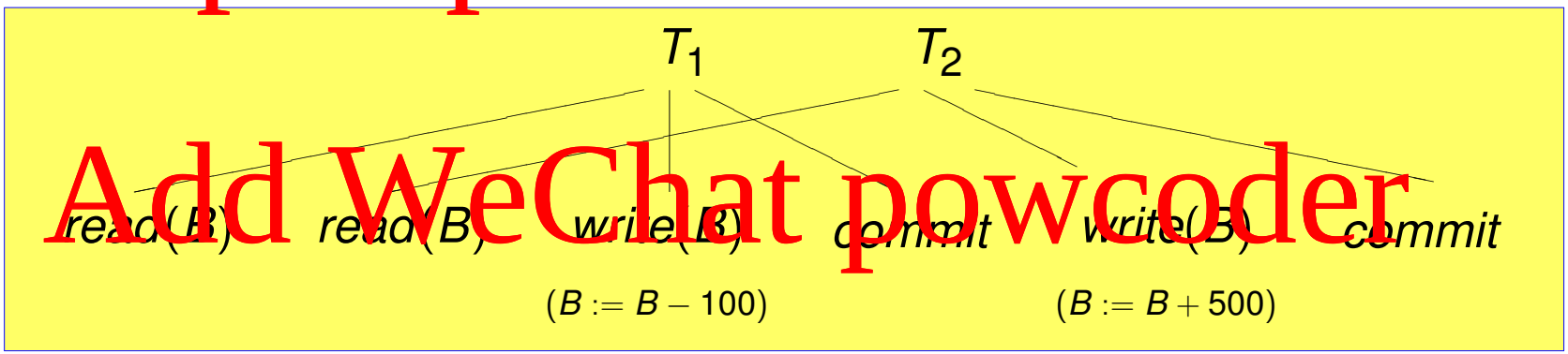
Steps	B(Bob)
before 1	$200
after 2	$200
after 4	$100
after 6	$700

- **Answer:** Bob's balance should be **$600**. The update by T_1 is lost!

(1) - The Lost Update Problem

- Occurs when two transactions update the same object, and one transaction could overwrite the value of the object which has already been updated by another transaction (**write-write conflicts**).

- **Example:**



- *write*(B) by T_2 overwrites B, and the update by T_1 is *lost*.

(2) - The Dirty Read Problem

- **Example:** Bob withdraws $100 from his account ($T_1$) while Alice deposits $500 into Bobs account ($T_2$).


```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: ABORT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```


Steps	T_1	T_2
1	read(B)	
2	write(B) (B:=B-100)	
3		read(B)
4	abort	
5		write(B) (B:=B+500)
6		commit

Steps	B(Bob)
before 1	$200
after 1	$200
after 2	$100
after 4	$200
after 6	$600

- **Question:** What is the problem?

(2) - The Dirty Read Problem

- **Example:** Bob withdraws $100 from his account ($T_1$) while Alice deposits $500 into Bobs account ($T_2$).


```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: ABORT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```

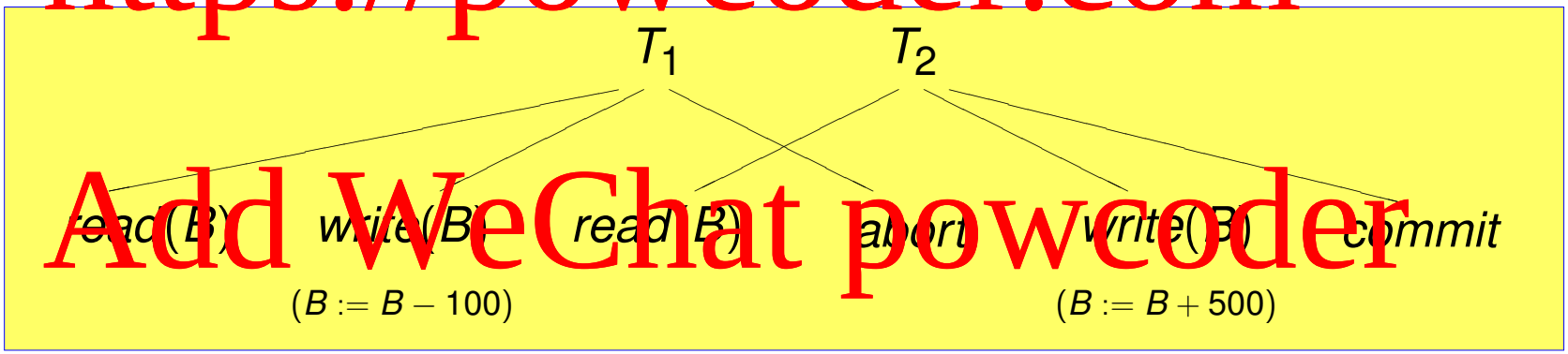
Steps	T_1	T_2
1	read(B)	
2	write(B) (B:=B-100)	
3		read(B)
4	abort	
5		write(B) (B:=B+500)
6		commit

Steps	B(Bob)
before 1	$200
after 1	$200
after 2	$100
after 4	$200
after 6	$600

- **Answer:** Bob's balance should be **$700** since T_1 was not completed.

(2) - The Dirty Read Problem

- Occurs when one transaction could read the value of an object that has been updated by another transaction but has not yet committed (**write-read conflicts**).
- **Example:**



- T_1 fails and must change the value of B back to **$200**; but T_2 has read the uncommitted ($\cong$ *dirty*) value of B (**$100**).

(3) - The Unrepeatable Read Problem

- **Example:** Bob checks his account (T_1) twice (takes time to decide whether to withdraw $200) while Alice withdraws $500 from Bob's account ($T_2$)


```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: UPDATE ACCOUNT SET balance=balance-500 WHERE name='Bob';
T2: COMMIT;
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
```


Steps	T_1	T_2
1	read(B)	
2		read(B)
3		write(B) (B:=B-500)
4		commit
5	read(B)	

Steps	B(Bob)
before 1	$500
after 2	$500
after 3	$0
after 4	$0
after 5	$0

- **Question:** What is the problem?

(3) - The Unrepeatable Read Problem

- **Example:** Bob checks his account (T_1) twice (takes time to decide whether to withdraw $200) while Alice withdraws $500 from Bob's account ($T_2$).

```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: UPDATE ACCOUNT SET balance=balance-500 WHERE name='Bob';
T2: COMMIT;
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
```


Steps	T_1	T_2
1	read(B)	
2		read(B)
3		write(B) (B:=B-500)
4		commit
5	read(B)	

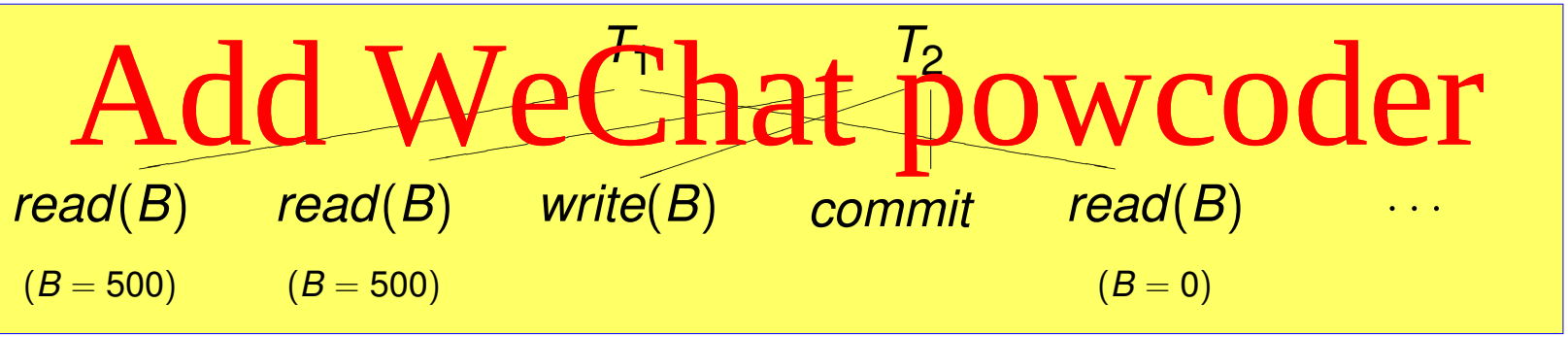
Steps	B(Bob)
before 1	$500
after 2	$500
after 3	$0
after 4	$0
after 5	$0

- **Answer:** Bob received two different account balances **$500** and **$0**, even though he hasn't withdrawn any money yet.

(3) - The Unrepeatable Read Problem

- A transaction could change the value of an object that has been read by another transaction but is still in progress (could issue two read for the object, or a write after reading the object) (**read-write conflicts**).

- **Example:**



(4) - The Phantom Read Problem

- **Example:** A query is submitted for finding all customers whose account balances are less than **$300** ($T_1$) while Alice is opening a new account with the balance **$200** ($T_2$).
- Assume that only Bob (B) has an account whose balance is less than **$300** before Alice (A) opens his new account.


```
T1: SELECT name FROM ACCOUNT WHERE balance<300;
T2: INSERT INTO ACCOUNT(id, name, balance) VALUES(99, 'Alice', 250);
T2: COMMIT;
T1: SELECT name FROM ACCOUNT WHERE balance<300;
```


Steps	T_1	T_2
1	read(R)	
2		write(R)
3		commit
4	read(R)	

Steps	Query result
before 1	$R = \{B\}$
after 1	$R = \{B\}$
after 2	$R = \{A, B\}$
after 4	$R = \{A, B\}$

- **Question:** What is the problem?

(4) - The Phantom Read Problem

- **Example:** A query is submitted for finding all customers whose account balances are less than **$300** ($T_1$) while Alice is opening a new account with the balance **$200** ($T_2$).
- Assume that only Bob (B) has an account whose balance is less than **$300** before Alice (A) opens his new account.


```
T1: SELECT name FROM Account WHERE balance<300;
T2: INSERT INTO Account(id, name, balance) VALUES(99, 'Alice', 250);
T2: COMMIT;
T1: SELECT name FROM Account WHERE balance<300;
```


Steps	T_1	T_2
1	read(R)	
2		write(R)
3		commit
4	read(R)	

Steps	Query result
before 1	$R = \{B\}$
after 1	$R = \{B\}$
after 2	$R = \{A, B\}$
after 4	$R = \{A, B\}$

- **Answer:** T_1 reads `Account` based on the condition `balance<300` twice but gets two different results $\{B\}$ and $\{A, B\}$.

(4) - The Phantom Read Problem

- Occurs when tuples updated by a transaction T_1 satisfy the search conditions of another transaction so that, by the same search condition, the transaction obtains different results at different times.

- **Example:**