Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION;
        UPDATE Bank SET amount = amount – 100
        WHERE name = 'Bob';
        UPDATE Bank SET amount = amount + 100
        WHERE name = 'Joe';
COMMIT;  (OR ROLLBACK;)
```

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

Recovery & Durability: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

Concurrency: Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation

1. Recovery & Durability of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full**, **or not at all**; keep log to be able to “roll-back” TXNs

Protection against crashes / aborts

Client 1:

```
INSERT INTO SmallProduct(name, price)
SELECT pname, price
FROM Product
WHERE price <= 0.99;

DELETE Product
WHERE price <= 0.99;
```

Crash / abort!

What goes wrong?

Protection against crashes / aborts

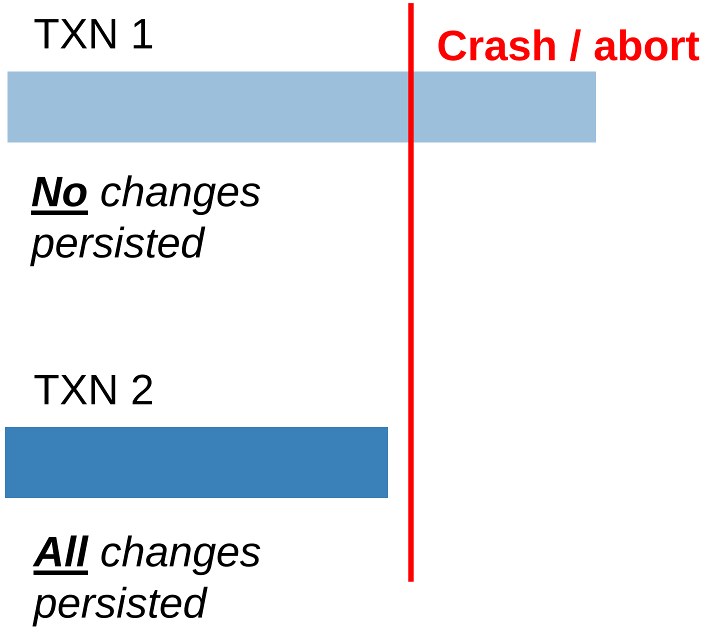
```
Client 1:
        START TRANSACTION;
                INSERT INTO SmallProduct(name, price)
                SELECT pname, price
                FROM Product
                WHERE price <= 0.99;

                DELETE Product
                WHERE price <= 0.99;
        COMMIT;
```

Atomicity & Durability

- **<u>A</u>**tomicity:
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen

- **<u>D</u>**urability:
  - If DBMS stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*



ACID: Atomicity

- TXN’s activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

ACID: Durability

- The effect of a TXN must continue to exist (***“persist”***) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…

- Means: Write data to **disk**

Motivation

<u>2. Concurrent</u> execution of user programs is essential for good DBMS performance.

- Individual TXNs might be *slow*- don't want to block other users during

- Disk access may be *slow*- let some TXNs use CPUs while others accessing disk

- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs humming…

Example- consider two TXNs:

```
T1: START TRANSACTION;
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = ‘A’;

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = ‘B’;
COMMIT;
```

```
T2: START TRANSACTION;
        UPDATE Accounts
        SET Amt = Amt * 1.06;
COMMIT;
```

T1 transfers $100 from B’s account to A’s account

T2 credits both accounts with a 6% interest payment

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule $\mathbf{T_1,T_2}$:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

Interleaved schedule A:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$106

Same result!

Scheduling examples

Starting Balance

A	B
\$50	\$200

Serial schedule $\mathbf{T_1,T_2}$:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
\$159	\$106

Different result than serial T_1,T_2!

Interleaved schedule B:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
\$159	\$112

Scheduling examples

Starting Balance

A	B
$50	$200

Serial schedule $\mathbf{T_2,T_1}$:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$153	$112

Interleaved schedule B:

T_1: A += 100 | B -= 100

T_2: A *= 1.06 | B *= 1.06

A	B
$159	$112

Different result than serial T_2,T_1 ALSO!

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency such that…

  Isolation is maintained: Users must be able to execute each TXN **as if they were the only user**

  - DBMS handles the details of *interleaving* various TXNs

  Consistency is maintained: TXNs must leave the DB in a **consistent state**

  - DBMS handles the details of enforcing integrity constraints

ACID: Consistency

- The tables must always satisfy user-specified ***integrity constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can’t be negative

- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

ACID: Isolation

- A transaction executes concurrently with other transactions
- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.
    - E.g. Should not be able to observe changes from other transactions during the run

Isolation Levels

- Fully serializable isolation is more expensive than “no isolation”
  - We can’t do as many things concurrently
- For performance, we generally want to specify the most relaxed *isolation level* that’s acceptable

Choosing the Isolation Level

- Before a transaction, we can say:

```
SET TRANSACTION ISOLATION LEVEL X
```

where *X* =

1. SERIALIZABLE (Default)
2. REPEATABLE READ
3. READ COMMITTED
4. READ UNCOMMITTED

- This affects what data the next transaction may see

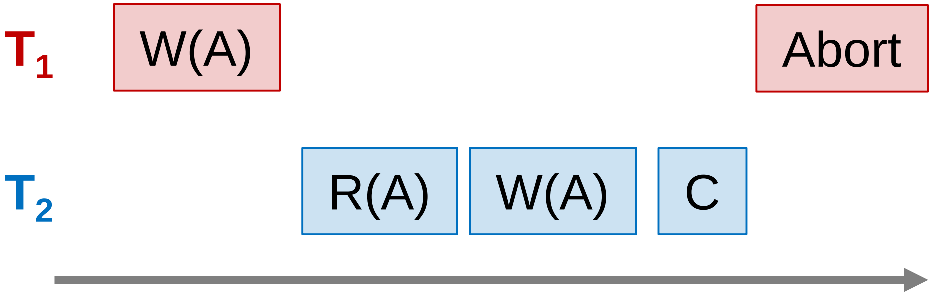
4 Read Uncommitted (T_2)

- A transaction running under READ UNCOMMITTED can see data in the database, even if it was written by a transaction that has not committed (and may never).

“Dirty read” / Reading uncommitted data:

Example:

allowed



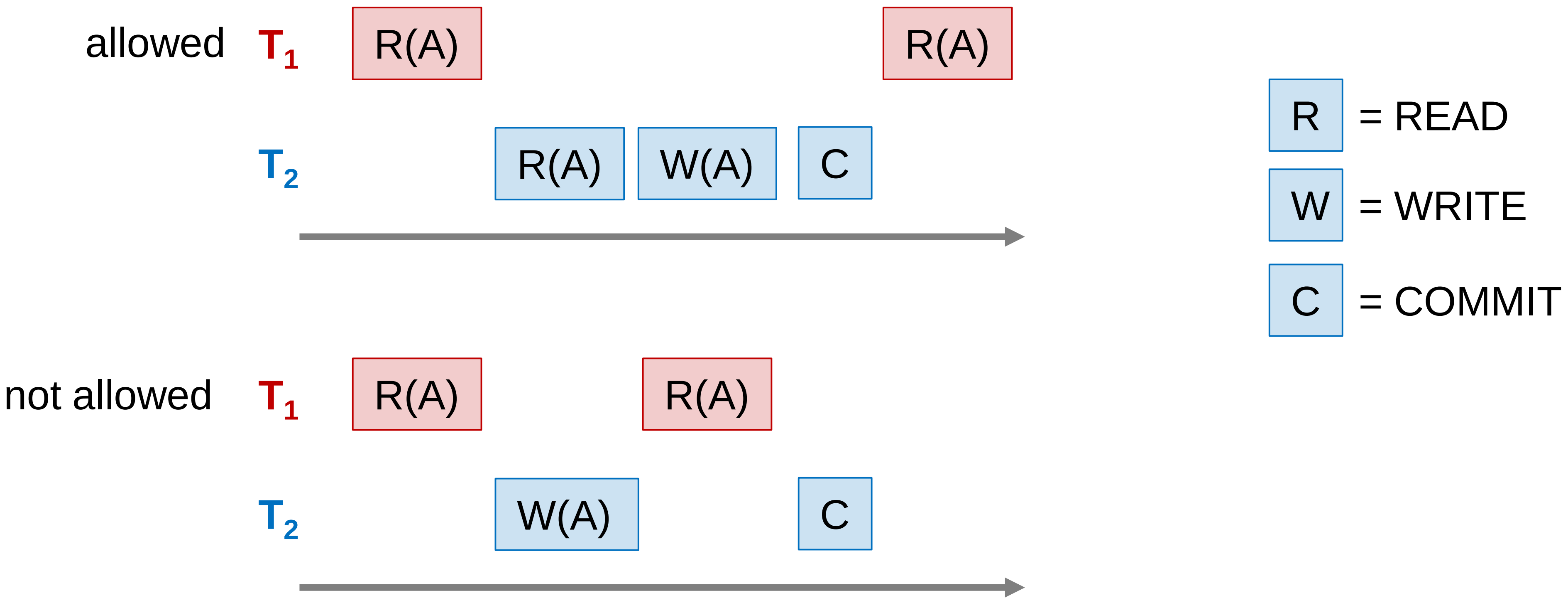
T_1 writes some data to A

T_2 reads from A, then writes back to A & commits

T_1 then aborts- *now T_2’s result is based on an obsolete / inconsistent value*

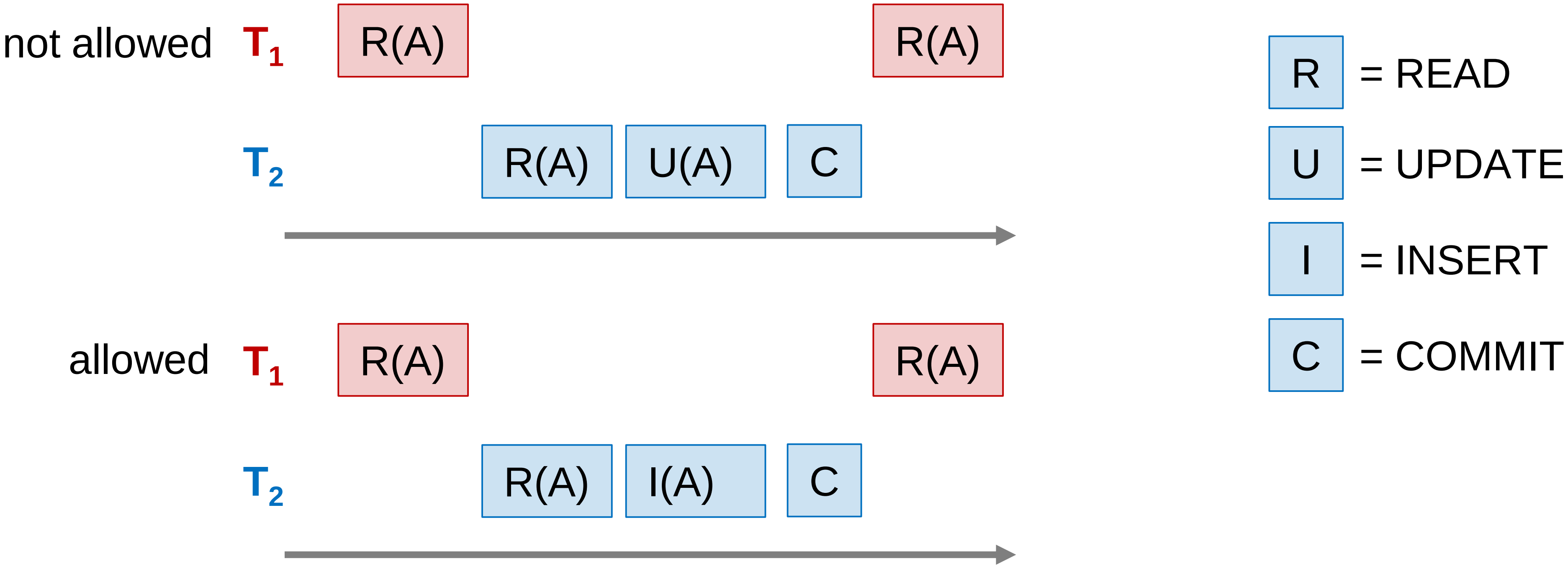
3 Read Commited (T_1)

- can see only committed data, but not necessarily the same data each time

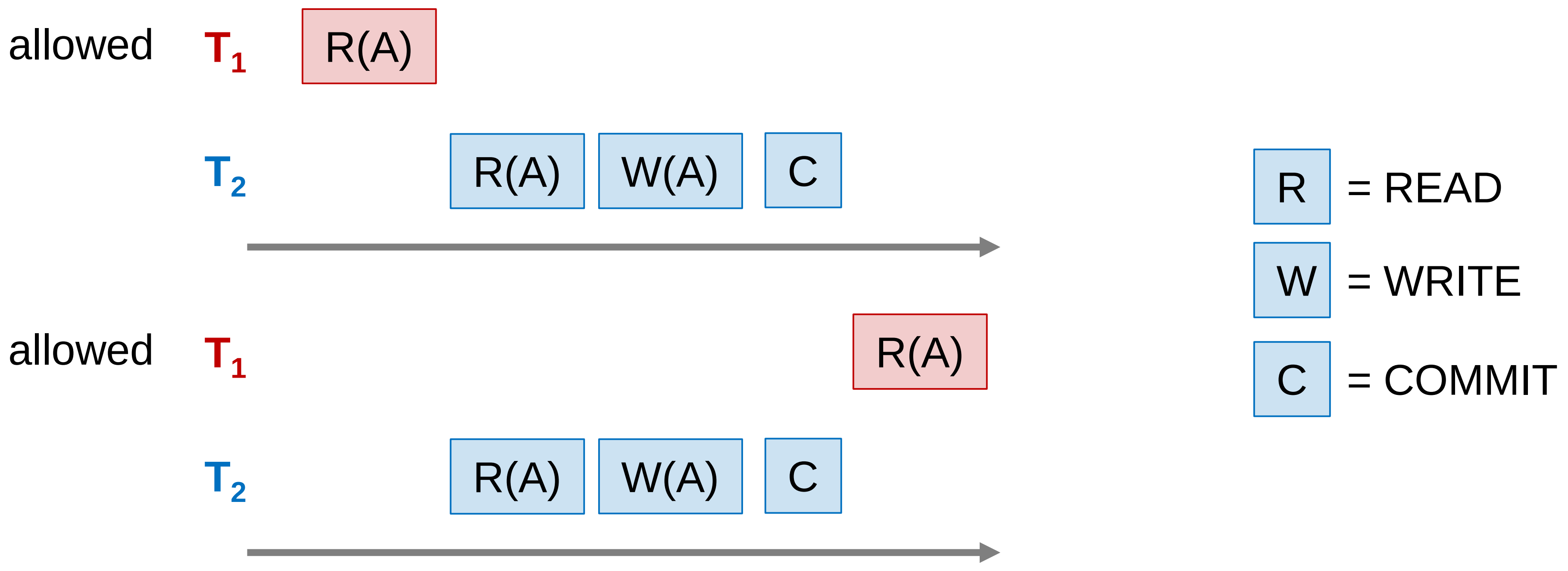


2 Repeatable Read (T_1)

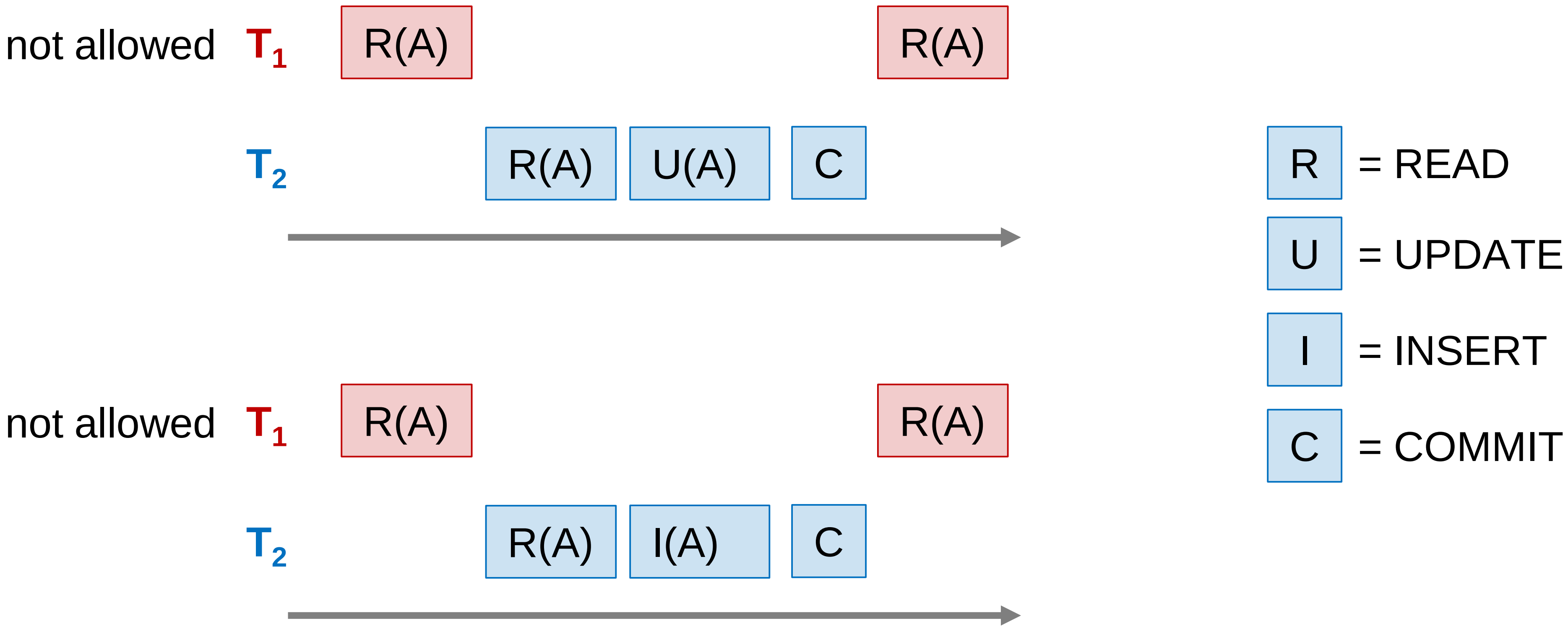
- Requirement is like read committed, plus: if data is read again, then everything seen the first time will be seen the second time.
  - But the second and subsequent reads may see *more* tuples as well.



1 Serializable (T_1)



1 Serializable (T_1)



Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions
- A and B are **equivalent schedules** if, ***for any database state***, the effect on DB of executing A **is identical to** the effect of executing B
- *A* **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

Scheduling examples

Interleaved schedule B:

T_1 A += 100 B -= 100

T_2 A *= 1.06 B *= 1.06

A	B
$159	$112

This schedule is different than ***any serial order!*** We say that it is **not serializable**

A	B
$159	$106

T_1, T_2

A	B
$153	$112

T_2, T_1

Serial Schedules

T1		A += 100	B -= 100		
T2				A *= 1.06	B*= 1.06

S1

T1				A += 100	B -= 100
T2		A *= 1.06	B *= 1.06		

S2

Interleaved Schedules

T1		A += 100		B -= 100	
T2			A *= 1.06		B*= 1.06

S3

T1			A += 100		B -= 100
T2		A *= 1.06		B *= 1.06	

S4

T1			A += 100	B -= 100	
T2		A *= 1.06			B*= 1.06

S5

T1		A += 100			B -= 100
T2			A *= 1.06	B *= 1.06	

S6

Serial Schedules	S1, S2
Serializable Schedules	S3, S4
Equivalent Schedules	<S1, S3> <S2, S4>
Non-serializable (Bad) Schedules	S5, S6

Conflict Types

Two actions of two different TXNs **<u>conflict</u>** if they involve the same variable, and at least one of them is a write

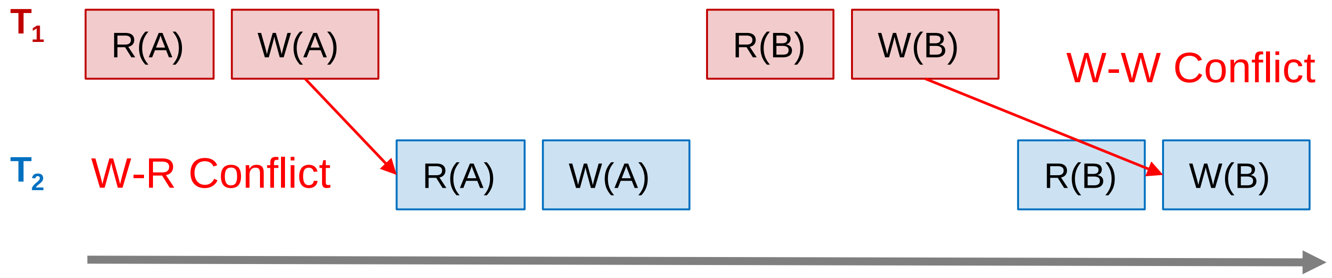
Thus, there are three types of conflicts:

- Read-Write conflicts (RW)
- Write-Read conflicts (WR)
- Write-Write conflicts (WW)

If we interchange the order of two conflicting actions, then the behavior of at least one TXN can change

Conflicts

Two actions of two different TXNs **<u>conflict</u>** if they involve the same variable, and at least one of them is a write



If we interchange the order of two conflicting actions, then the behavior of at least one TXN can change

Conflict Serializability

Two schedules are ***conflict equivalent*** if:

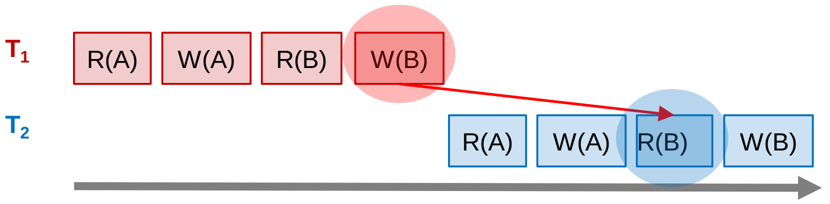
- They involve *the same actions of the same TXNs*
- Every *pair of conflicting actions* of two TXNs are *ordered in the same way*

Schedule S is ***conflict serializable*** if S is *conflict equivalent* to ***some*** serial schedule

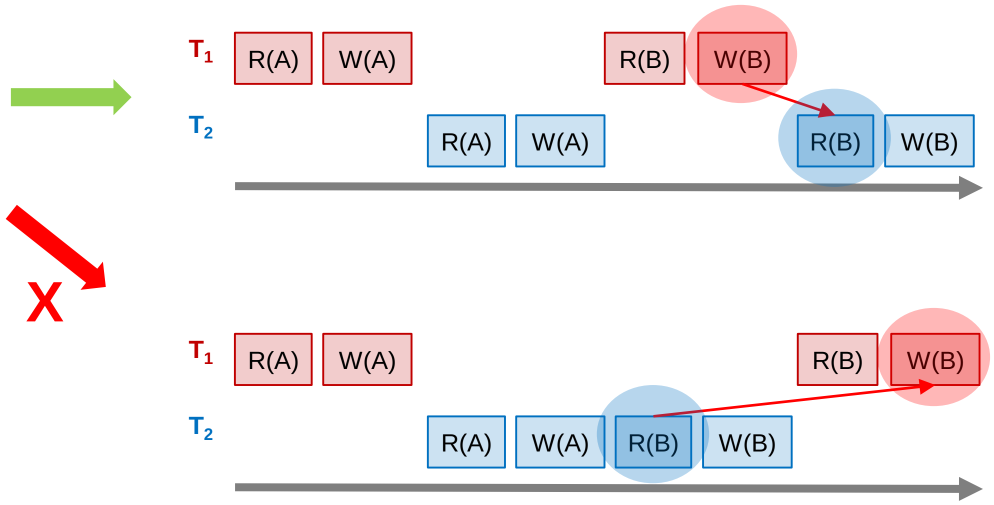
Conflict serializable ⇒ serializable
So if we have conflict serializable, we have consistency & isolation!

Example "Good" vs. "bad" schedules

Serial Schedule:



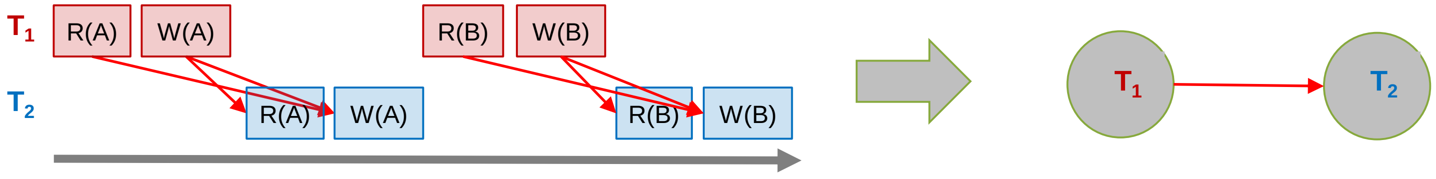
Interleaved Schedules:



Note that in the "bad" schedule, the ***order of conflicting actions is different than the above (or any) serial schedule!***

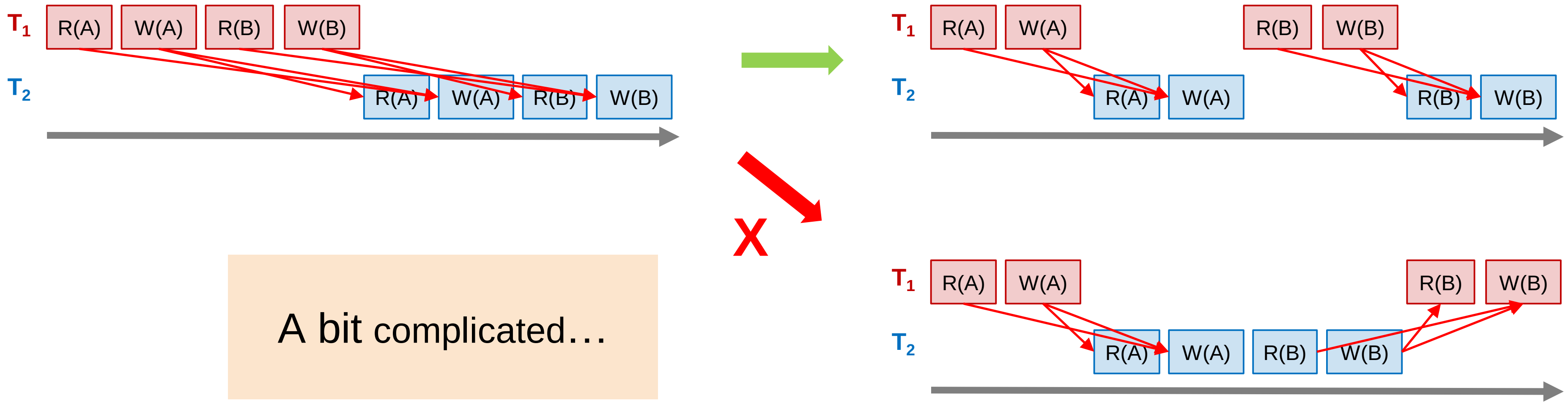
The Conflict Graph

- Let's now consider looking at conflicts **at the TXN level**
- Consider a graph where the **nodes are TXNs**, and there is an edge from $T_i \rightarrow T_j$ **if any actions in T_i <u>precede and conflict with</u> any actions in T_j**



What can we say about “good” vs. “bad” conflict graphs?

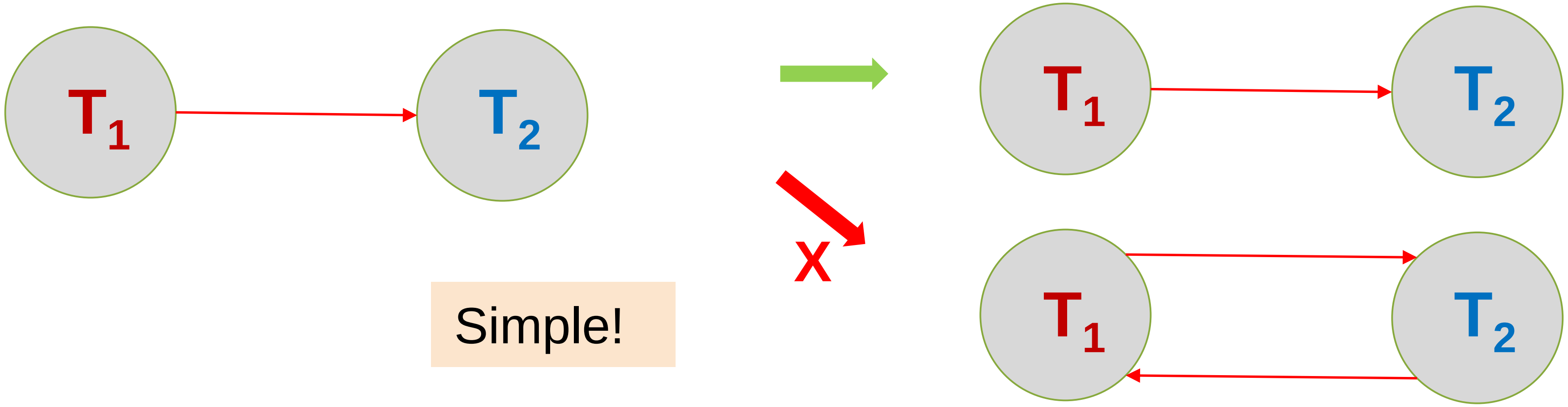
Serial Schedule: ***Interleaved Schedules***:



What can we say about “good” vs. “bad” conflict graphs?

Serial Schedule:

Interleaved Schedules:



Theorem: Schedule is **conflict serializable** if and only if its conflict graph is **acyclic**