

# Database Usage (and Construction)

## Transactions


# Setting

- DBMS must allow concurrent access to databases.
  - Imagine a bank where account information is stored in a database *not* allowing concurrent access. Then only one person could do a withdrawal in an ATM machine at the time – anywhere!
- Uncontrolled concurrent access may lead to problems.


## Example:

Imagine a program that does the following:

1. Get a day, a time and a course from the user in order to schedule a lecture. (**get**)
2. List all available rooms at that time, with number of seats, and let the user choose one. (**list**)
3. Book the chosen room for the given course at the given time. (**book**)



```
SELECT *  
FROM ROOMS  
WHERE name NOT IN  
  (SELECT room  
   FROM Lectures  
   WHERE weekday = theDay  
    AND hour = theTime);
```



```
INSERT INTO Lectures VALUES  
(theCourse, thePeriod,  
 theDay, theTime,  
 chosenRoom);
```

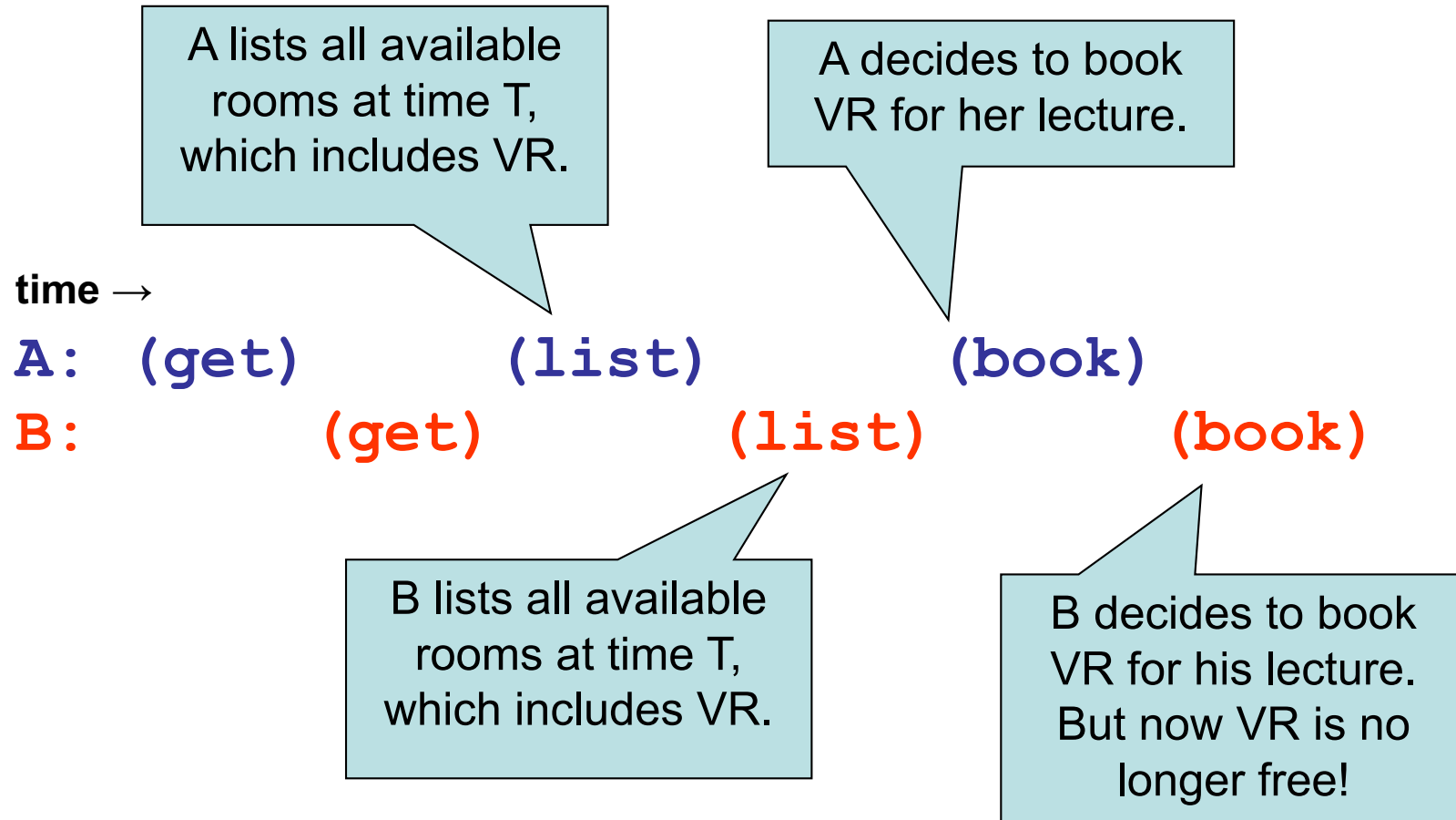
# Running in parallel

- Assume two people, A and B, both try to book a room for the same time, at the same time.
- Both programs perform the sequence **(get)** **(list)** **(book)**, in that order.
- But we can interleave the blocks of the two sequences in any way we like!
  - Here's one possible interleaving:

A: **(get)**                    **(list)**                    **(book)**

B:                    **(get)**                    **(list)**                    **(book)**

# Interleaving



# Quiz!

Look again at the interleaving:

time →

A: (get)                    (list)                    (book)

B:                    (get)                    (list)                    (book)

What can we do to fix it?

The only way that we get the desired behavior is if both A and B may perform the operations `(list)` `(book)` without the other doing a `(book)` in the middle!

# Quiz!

Assume we run the following two programs in parallel, and assume the databases contains only the Databases lecture in room VR on Mondays (and all lectures are 2h long):

$P_1$ :

1. Insert a lecture for the Databases course in room VR at 10 on Mondays.  
(**ins**)
2. Delete the lecture in the Databases course in room VR at 13 on Mondays.  
(**del**)

$P_2$ :

1. Find the first lecture of the day in room VR on Mondays.  
(**min**)
2. Find the last lecture of the day in room VR on Mondays.  
(**max**)
3. Return the total time that room VR is occupied,  
( $(max+2)-min$ ). (**ret**)

# ...Quiz continued!

|       |                   |                          |
|-------|-------------------|--------------------------|
| $P_1$ | (ins) (del)       | What could $P_2$ return? |
| $P_2$ | (min) (max) (ret) |                          |

- Need to consider possible **schedules** of the actions that access or update the database: (ins) (del) (min) (max)

(ins) (del) (min) (max)

$P_2$  returns 2

(ins) (min) (del) (max)

$P_2$  returns 2

(ins) (min) (max) (del)

$P_2$  returns 5

(min) (ins) (del) (max)

$P_2$  returns -1

(min) (ins) (max) (del)

$P_2$  returns 2

(min) (max) (ins) (del)

$P_2$  returns 2



# Serializability

- Two programs are run *in serial* if one finishes before the other starts.
- The running of two programs is *serializable* if the effects are the same as if they had been run in serial.

|    |       |        |        |
|----|-------|--------|--------|
| A: | (get) | (list) | (book) |
| B: | (get) | (list) | (book) |

Not serializable

Serializable

|    |       |        |        |
|----|-------|--------|--------|
| A: | (get) | (list) | (book) |
| B: | (get) | (list) | (book) |

## Example:

Assume we perform the following operations to transfer 100 SEK from account X to account Y.

1. Check account balance in account X.



```
SELECT balance
FROM Accounts
WHERE accountID = X;
```

2. Subtract 100 from account X.



```
UPDATE Accounts
SET balance = balance - 100
WHERE accountID = X;
```

3. Add 100 to account Y.



```
UPDATE Accounts
SET balance = balance + 100
WHERE accountID = Y;
```

**What could go wrong?**

## Example:

Assume we perform the following operations to transfer 100 SEK from account X to account Y.

1. Check account balance in account X.



```
SELECT balance
FROM Accounts
WHERE accountID = X;
```

2. Subtract 100 from account X.



```
UPDATE Accounts
SET balance = balance - 100
WHERE accountID = X;
```

3. Add 100 to account Y.



```
UPDATE Accounts
SET balance = balance + 100
WHERE accountID = Y;
```

Two things can go wrong: We can have strange interleavings like before. But also, assume the program crashes after executing 1 and 2 – we'll have lost 100 SEK!

# Atomicity

- For many programs, we require that "all or nothing" is executed.
  - We say a sequence of actions is executed *atomically* if it is executed either in entirety, or not at all.
    - The state in the middle is never visible from outside the sequence.
    - cf. Greek atom = indivisible.
    - In case of a crash in the middle, any changes that were made up until that point must be undone.

# ACID Transactions

- A DBMS is expected to support "ACID transactions", which are
  - **Atomic**: Either the whole transaction is run, or nothing.
  - **Consistent**: Database constraints are preserved.
  - **Isolated**: Different transactions may not interact with each other.
  - **Durable**: Effects of a transaction are not lost in case of a system crash.

# Transactions in SQL

- SQL supports transactions, often behind the scenes.
  - An SQL statement is a transaction.
    - E.g. an update of a table can't be interrupted after half the rows.
    - Any triggers, procedures, functions etc. that are started by the statement is part of the same transaction.

# Controlling transactions

- We can explicitly start transactions using the **START TRANSACTION** or **BEGIN** statement, and end them using **COMMIT** or **ROLLBACK**:
  - **COMMIT** causes an SQL transaction to complete successfully.
    - Any modifications done by the transaction are now permanent in the database.
  - **ROLLBACK** or **ABORT** causes an SQL transaction to end by aborting it.
    - Any modifications to the database must be undone.
    - Rollbacks could be caused implicitly by errors e.g. division by 0.

# Read-only vs. Read-write

- A transaction that does not modify the database is called *read-only*.
  - A read-only transaction can never interfere with another transaction (but not the other way around!).
  - Any number of read-only transactions can be run concurrently.
- A transaction that both reads and modifies the database is called *read-write*.
  - No other transaction may write between the read and write.



# SET TRANSACTION

- We can hint the DBMS that a transaction only does reading, by issuing the statement:

**SET TRANSACTION READ ONLY;**

- Possibly the DBMS can make use of the information and optimize scheduling.

# Drawbacks

- Serializability and atomicity are necessary, but don't come without a cost.
  - We must retain old data until the transaction commits.
  - Other transactions may need to wait for one to complete.
- In some cases some interference may be acceptable, and could speed up the system greatly.

## Example:

Recall the first example of booking rooms:

time →

A: (get)                    (list)                    (book)

B:                    (get)                    (list)                    (book)

It could take time for the user to decide which room to choose after getting the list. If we make this a serializable transaction, all other users would have to wait as well.

The worst thing that could happen is that B is told to choose another room when he tries to book the room that A just booked.

# Isolation levels

- ANSI SQL standard defines four *isolation levels*, which are choices about what kinds of interference are allowed between transactions.
- Each transaction chooses its own isolation level, deciding how other transactions may interfere with it.
- Isolation level is defined in terms of three phenomena that can occur.

# Kinds of interference

The ANSI SQL standard describes:

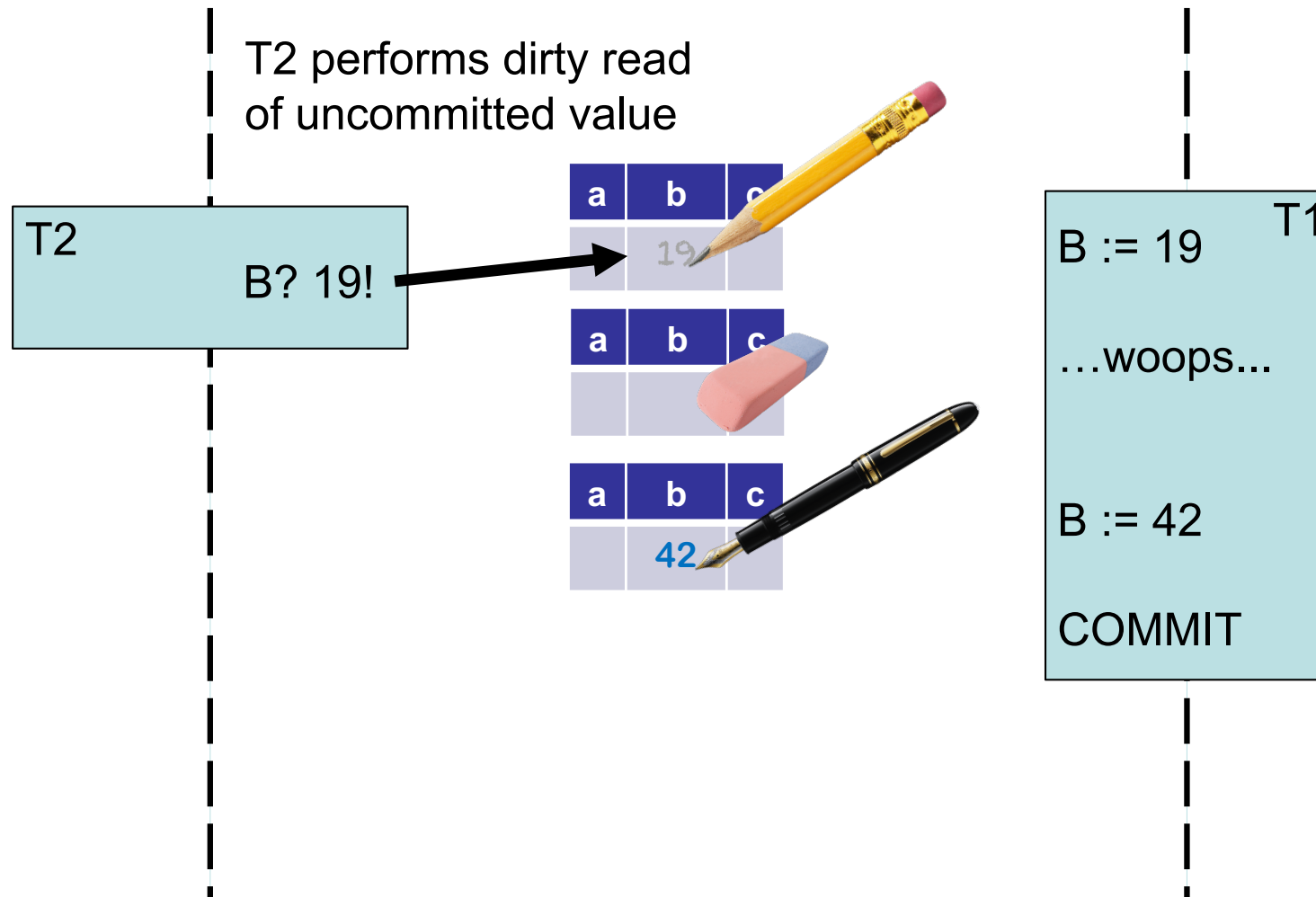
- Dirty read
- Non-repeatable read
- Phantom

(These, and other kinds of interference, are discussed in: Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., & O'Neil, P. (1995). A critique of ANSI SQL isolation levels. ACM SIGMOD Record, 24(2), 1-10.)

# Dirty read

- Transaction T1 modifies a data item.
- Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK.
- If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed.

# Dirty read example

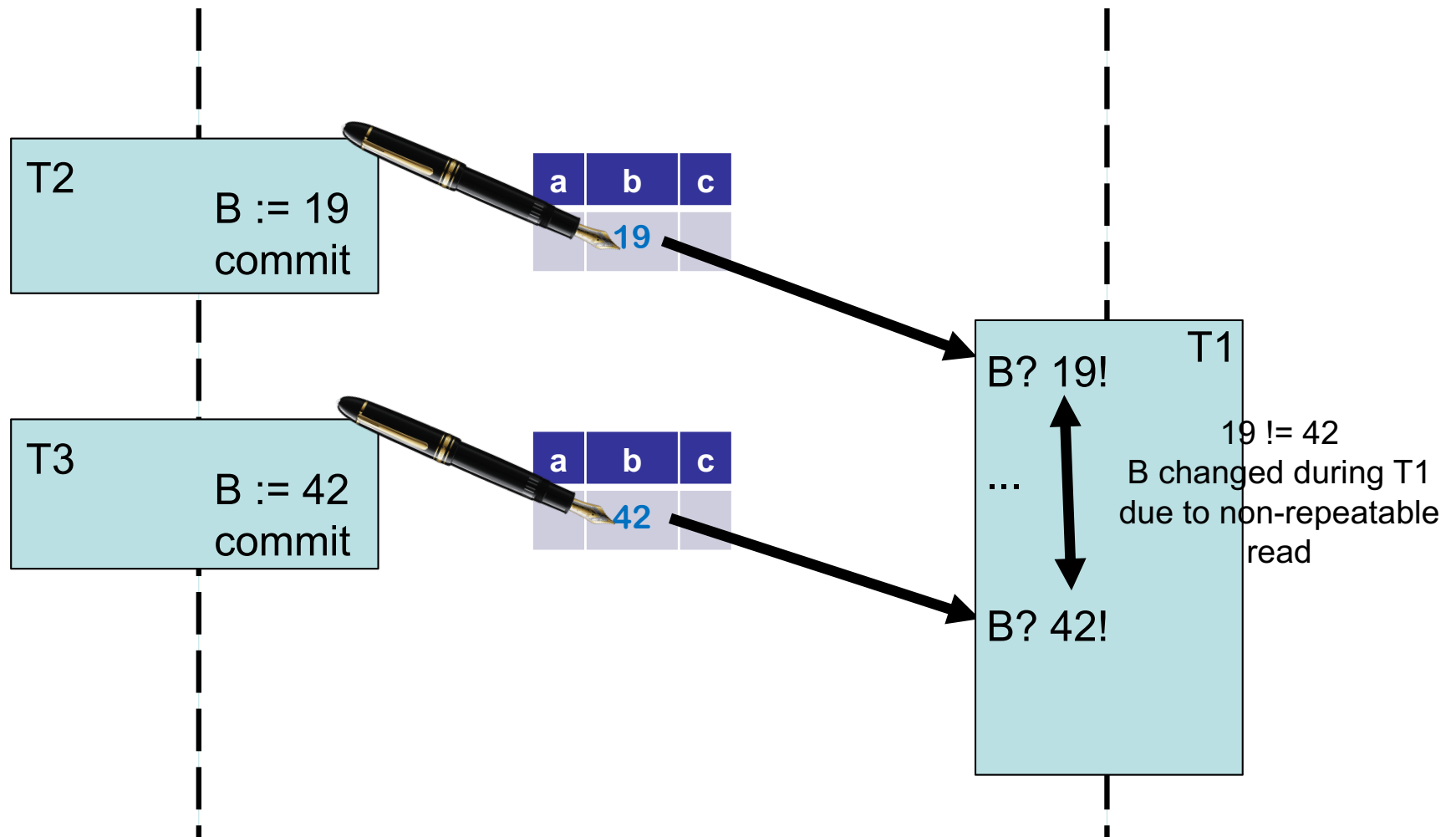


# Non-repeatable read

- Transaction T1 reads a data item.
- Another transaction T2 then modifies or deletes that data item and commits.
- If T1 then attempts to re-read the data item, it receives a modified value or discovers that the data item has been deleted.



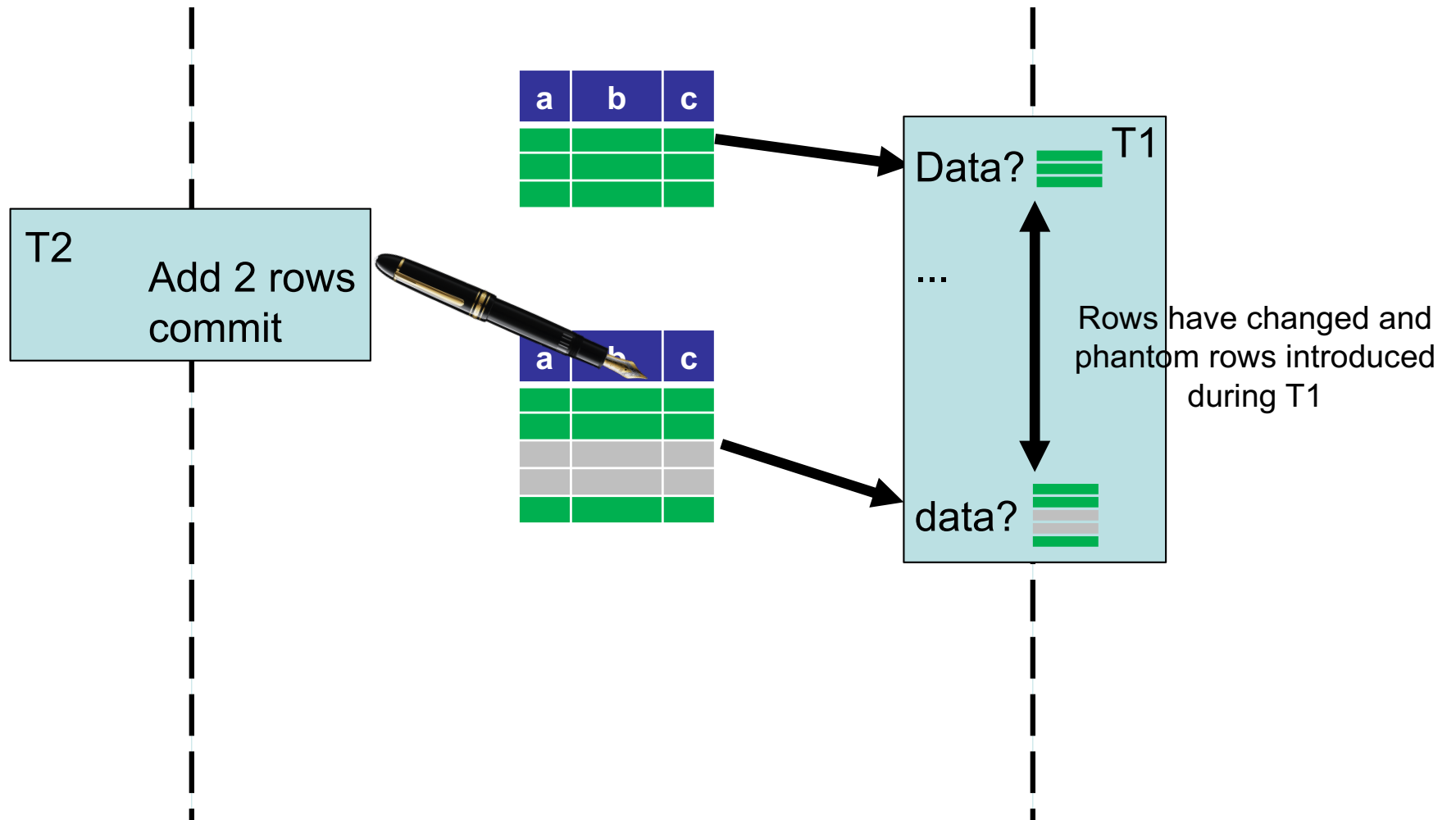
# Non-repeatable read example



# Phantom

- Transaction T1 reads a set of data items satisfying some <search condition>.
- Transaction T2 then creates data items that satisfy T1's <search condition> and commits.
- If T1 then repeats its read with the same <search condition>, it gets a set of data items different from the first read.

# Phantom example



# Choosing isolation level

- Within a transaction we can choose the isolation level:

**SET TRANSACTION ISOLATION LEVEL X;**

where X is one of

- **SERIALIZABLE**
- **READ COMMITTED**
- **READ UNCOMMITTED**
- **REPEATABLE READ**

# Isolation levels - differences

What kinds of interference are possible?

|                  | Dirty reads | Non-repeatable reads | Phantoms |
|------------------|-------------|----------------------|----------|
| READ UNCOMMITTED | Yes         | Yes                  | Yes      |
| READ COMMITTED   | No          | Yes                  | Yes      |
| REPEATABLE READ  | No          | No                   | Yes      |
| SERIALIZABLE     | No          | No                   | No       |



Increasing  
Isolation  
strictness

# READ UNCOMMITTED

- If a transaction is run with isolation level **READ UNCOMMITTED**, then the transaction allows other transactions to modify the database while running.
- Anything that is changed by another transaction affects the reads of this transaction, even if the other transaction has not yet committed!

# READ COMMITTED

- If a transaction is run with isolation level **READ COMMITTED**, then the transaction allows other transactions to modify the database while running.
- Anything that is committed by another transaction affects the reads of this transaction.

# REPEATABLE READ

- If a transaction is run with isolation level **REPEATABLE READ**, it works like read committed, except:
- If the transaction reads more than once, we are guaranteed to get *at least* the same tuples again (though we could get more).



# SERIALIZABLE

- If a transaction is run with isolation level **SERIALIZABLE**, then no other transaction may interfere with it in any way.
  - Examples:

If two room booking transactions are run serializable, then a booking for a room that was listed as free will always succeed, and transactions must wait for other transactions to finish.

In the min-max example, we always get a value that is correct at some point in time, either before or after the updating.

# Quiz!

If we extend the room booking transaction with a confirmation, i.e. **(list) (book) (confirm)**, and run two in parallel with isolation level **READ UNCOMMITTED**, what could happen?

Same as with **READ COMMITTED**, except that if the user of the first transaction changes her mind at confirmation, thus causing a roll-back, the second user could be told that the room is booked even though it never was!

# Quiz!

If we run two room booking transactions, **(list)** **(book)**, in parallel with isolation level **READ COMMITTED**, what would happen?

One transaction could book a room after the other had listed it as free, and the second booking may fail.

On the other hand, no transaction must wait for any other to finish.

# Quiz!

If we run two room booking transactions, **(list)** **(book)**, in parallel with isolation level **REPEATABLE READ**, what would happen?

Exactly the same thing as for **READ COMMITTED**, since we only read once!

# Quiz again!

If we run the first transactions of the min-max example as **READ UNCOMMITTED**, what could happen?

The update could be done between **(min)** and **(max)**, which means we could get the value -1.

Even if the updating is run **SERIALIZABLE**, we could see the state between **(ins)** and **(del)**, so the value 5 is also possible in this case!

Remember: Isolation level is a personal choice. Only because the min-max transaction is read-only can we run it in the middle of a serializable transaction!

# Quiz again!

If we run the first transactions of the min-max example, ( **min** ) ( **max** ) and ( **ins** ) ( **del** ), as **READ COMMITTED**, what could happen?

The update could be done between min and max, which means we could get the value -1.

If the updating is run **SERIALIZABLE**, we could not see the state between since the changes would not be committed, so the value 5 is not possible.

# Quiz again!

If we run the first transactions of the min-max example as **REPEATABLE READ**, what could happen?

If the update is done between **(min)** and **(max)**, we will still see the deleted value when doing **(max)**, so we can only get the value 2.

... but if we do **(max)(min)** instead, we would get the value 5...

# Summary transactions

- DBMS must ensure that different processes don't interfere with each other!
  - "ACID": Atomicity, Consistency, Isolation, Durability.
  - The isolation levels of transactions may vary.
    - Serializable
    - Read Committed
    - Read Uncommitted
    - Repeatable Read
  - Isolation level affects only that transaction!



Next time,  
Database Optimization:

Indexes

Non-natural keys

Denormalization