# Transactions

Niklas Fors, EDAF20, 2019-02-05

*mostly based on "Designing Data-Intensive Applications" (chapter 7)*
*by Martin Kleppmann*

## Transactions

**Transaction**: a group of operations that are performed together

Many DBMSs have **ACID** support for transactions:

- **Atomicity** – either all operations are performed (committed), or none
- **Consistency** – keeps the database in a consistent state
- **Isolation** – concurrent transactions are isolated from each other
- **Duration** – committed transactions are permanently recorded

(What this means exactly varies a bit between DBMS…)

## Example

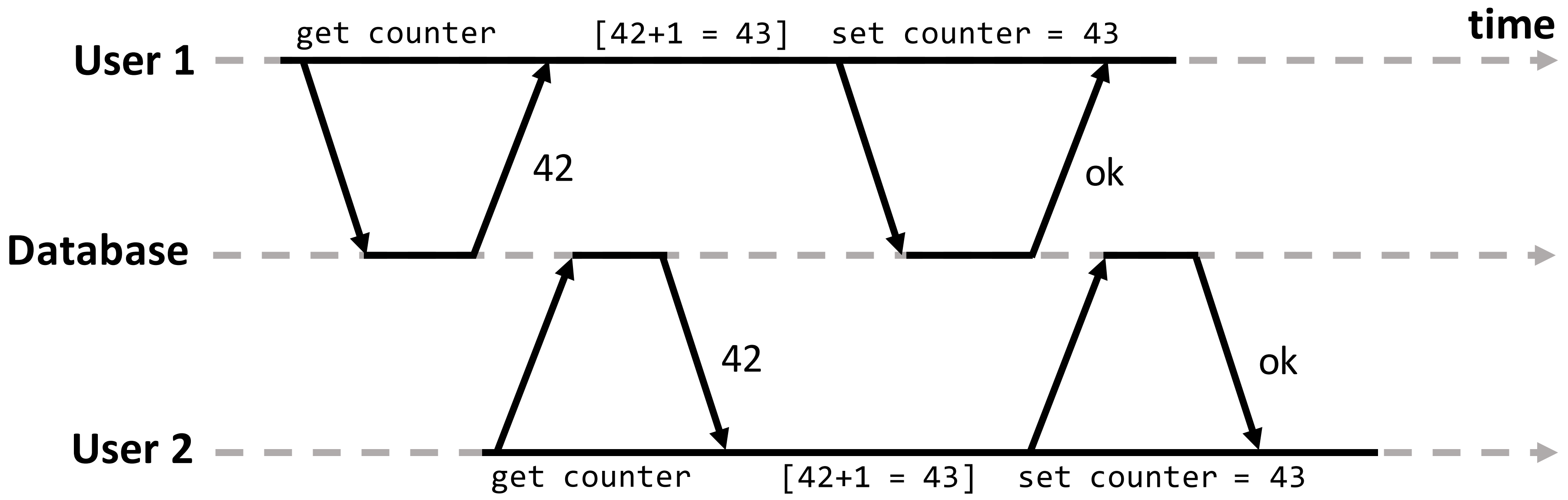
A transaction is started with a command (START TRANSACTION) and ended with another command (COMMIT to save changes, or ROLLBACK to undo all changes).

```
start transaction;
insert into A values(1);
insert into A values(2);
commit;
-- A contains values 1 and 2

start transaction;
insert into A values(3);
rollback;
-- A still contains 1 and 2
```

Normally, a client executes in ”auto-commit” mode. This means that each SQL statement is its own transaction — the changes performed by the statement are immediately committed (written to the database).

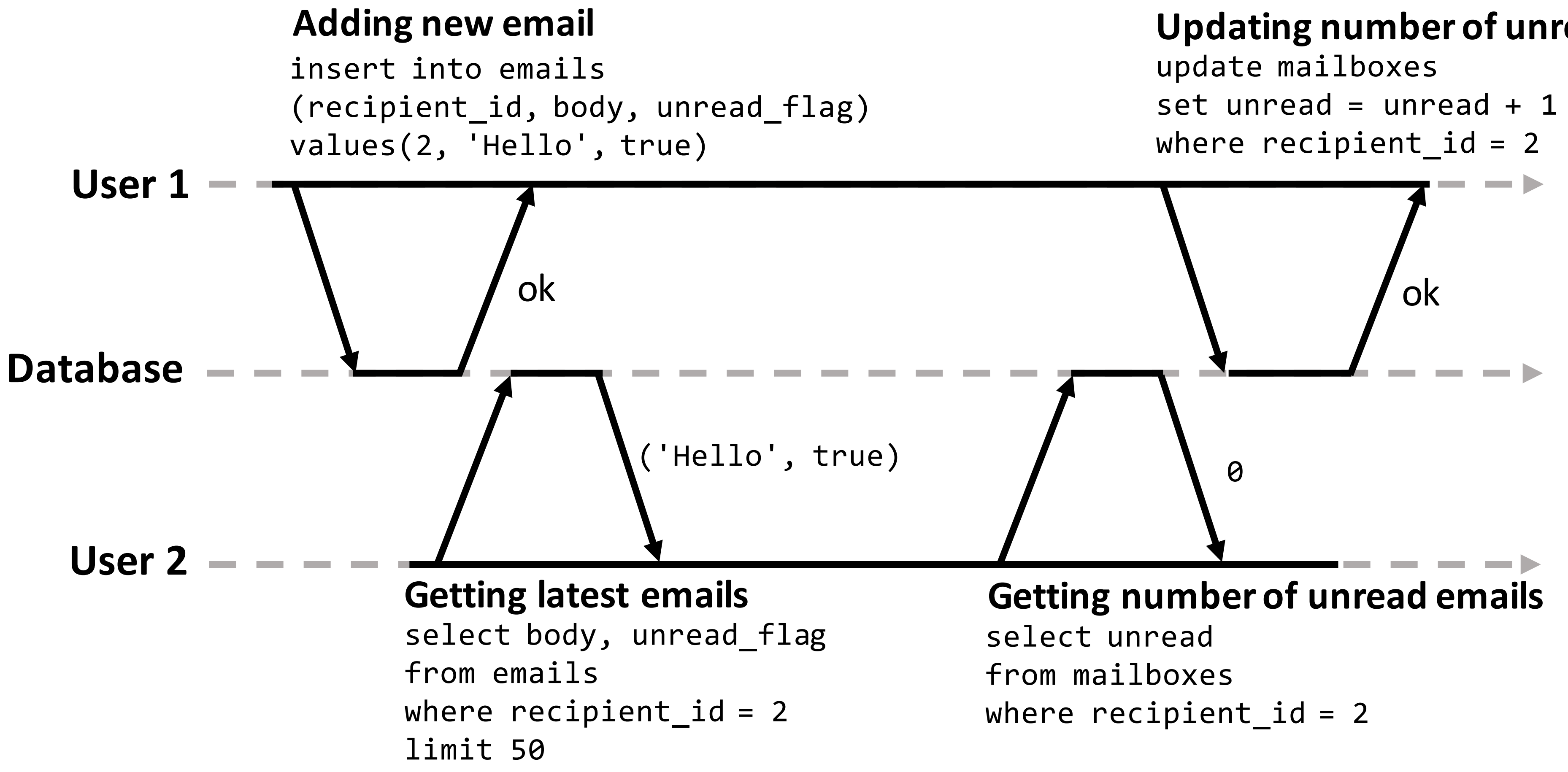
# Concurrency Issue – Lost Update



The counter is only incremented once!

Thus, one update is lost.

# Issue: Dirty Read



```
insert into emails
(recipient_id, body, unread_flag)
values(2, 'Hello', true)
```

```
update mailboxes
set unread = unread + 1
where recipient_id = 2
```

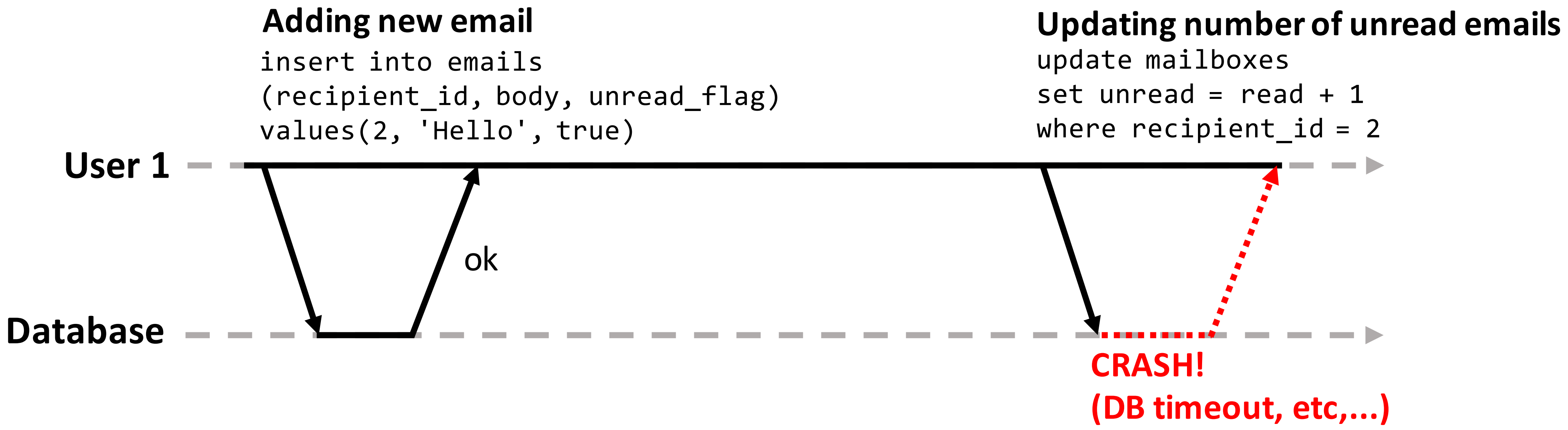
```
select body, unread_flag
from emails
where recipient_id = 2
limit 50
```

```
select unread
from mailboxes
where recipient_id = 2
```

User 2 reads an uncommitted value!

*User 2 retrieves the unread email, but not the correct number of unread emails, since the update occurs after the second select.*

# Atomicity



```
insert into emails
(recipient_id, body, unread_flag)
values(2, 'Hello', true)
```

```
update mailboxes
set unread = read + 1
where recipient_id = 2
```

If the second operation fails for some reason, we don’t want the first operation to be committed.

Thus, we want:
**atomicity** - either all operations are performed (committed), or none

## Isolation Levels – How Much Is Isolated?

Normally, the DBMS supports different *isolation levels*:

- **Read uncommitted**
  - Uncommitted values can be read
- **Read committed**
  - Only shows committed values
  - Prevents dirty reads
- **Repeatable read/snapshot isolation** (default in MySQL)
  - Shows a snapshot of the database
  - Prevents dirty reads and unrepeatable reads
- **Serializable**
  - Strongest guarantees (prevents all concurrency issues), but might be slow
  - Same effect as running transactions in serial

## Setting Isolation Level (SQL)

Set isolation level for the current session:

```
> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

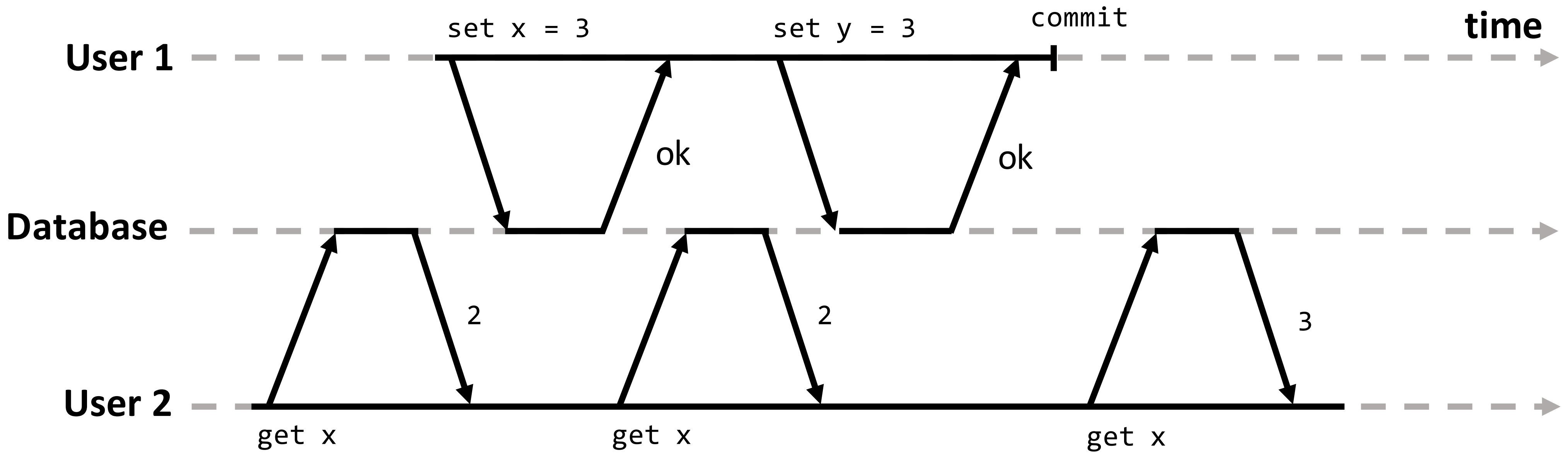
Allowed values:

```
READ UNCOMMITTED
READ COMMITTED
REPEATABLE READ (standard in MySQL)
SERIALIZABLE
```

Get current session isolation level in MySQL (v8):

```
> select @@transaction_isolation;
```

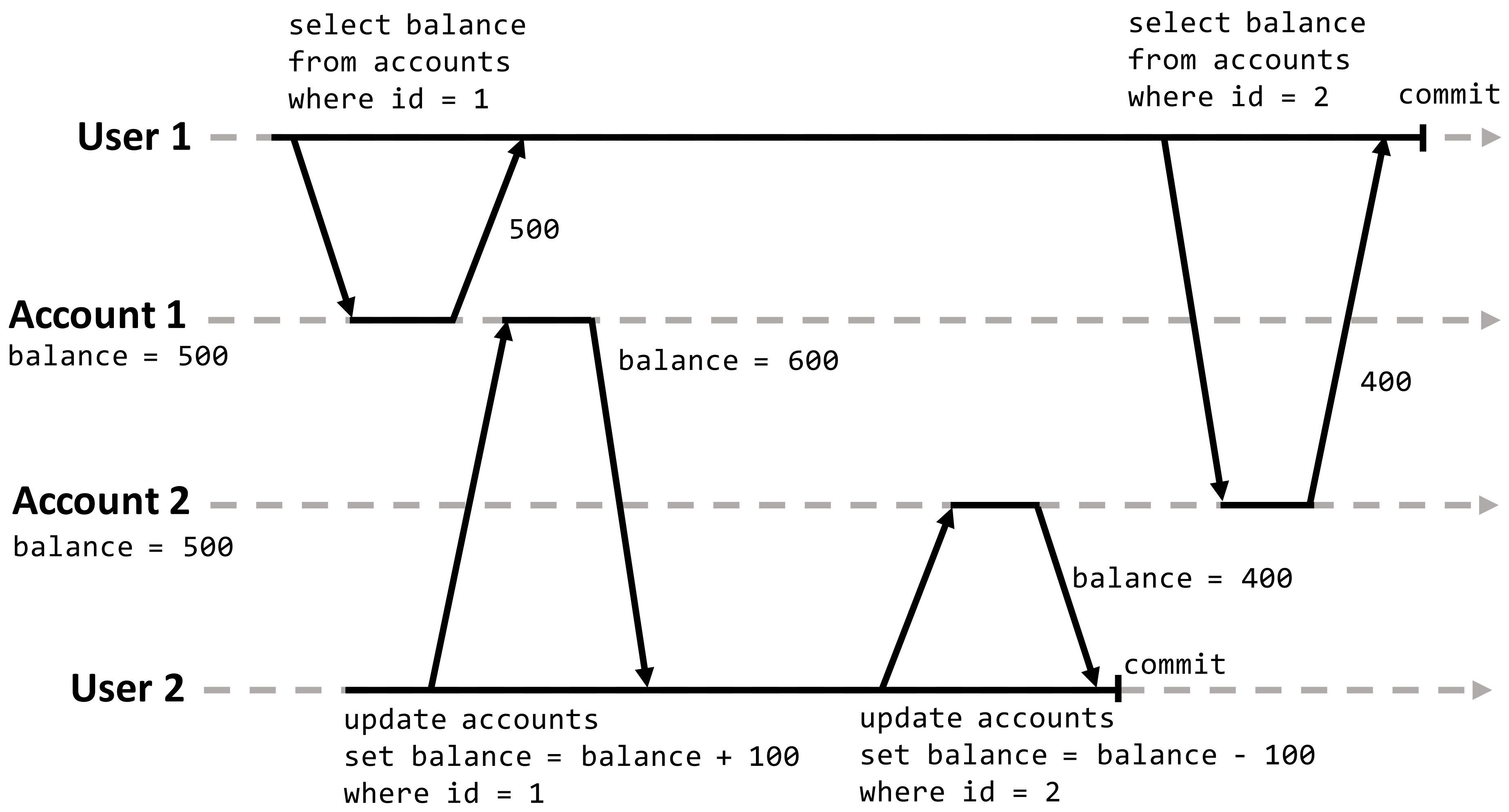
# Isolation Level: Read Committed



User 2 only reads committed values!

No dirty reads!

## Issue: Read Skew/Unrepeatable Read



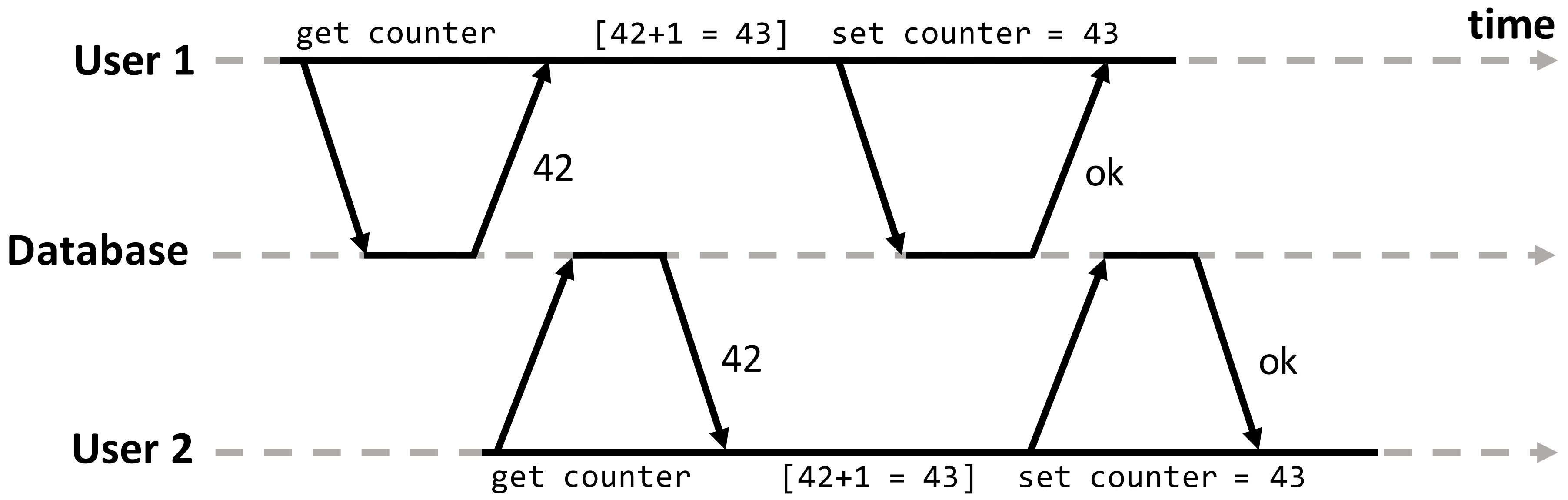
User 1 sees the value of Account 1 before User 2 commits and the value of Account 2 after the commit.
=> Sum: 500 + 400 = 900
(and not 1000!)

User 2 moves 100 from Account 2 to Account 1.

## Repeatable Read/Snapshot Isolation

- The **Read skew/Unrepeatable Read** issue is solved using the isolation level **Repeatable Read/Snapshot Isolation**.
- This isolation level normally creates a *snapshot* of the database when a new transaction starts, and no modifications by other transactions can be seen during the transaction.
- Normally implemented using *Multi-Version Concurrency Control* (MVCC)

# Back to Lost Updates



**Pattern:**
*read-modify-write*

**Lost Updates** are *not* solved using the isolation level **repeatable read.**

# Solutions to Lost Update

- Atomic write operations:
```
UPDATE counters SET value = value + 1 WHERE id = '1'
```
- Explicit locking (see next slide)
- Automatic detection of lost updates
  - Not available in MySQL, but available for the isolation level Repeatable Read in PostgreSQL, …
- Compare-and-set
```
UPDATE wiki_pages SET content = 'new content'
WHERE id = 1234 AND content = 'old content’
```

Use the old content to verify that the content hasn’t been changed by another transaction.

## Explicit Locking

```
START TRANSACTION;
SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE;

-- Check whether move is valid, then update the position
-- of the piece that was returned by the previous SELECT.

UPDATE figures SET position = 'c4' WHERE id = 1234;
COMMIT;
```

The FOR UPDATE locks the rows returned by the SELECT statement, and tells the database that these rows are going to be changed. The returned rows cannot be used by another transaction (if they also use locks).

# Locks in MySQL (InnoDB)

Default locking:

- SELECT statements set no locks
- UPDATE/INSERT/DELETE statements set write locks

If a SELECT is followed by a later update of the same item, it is necessary to explicitly set a lock on the item. This can be done by:

- `SELECT … LOCK IN SHARE MODE (read lock)`
- `SELECT … FOR UPDATE (write lock)`

## Another Example of Lock Usage

```
start transaction;
SELECT free FROM flights WHERE flight = 'A1' FOR UPDATE;
if (free == 0) {
  rollback;
}
UPDATE flights SET free = free – 1 WHERE flight = 'A1';
INSERT INTO tickets VALUES(…);
commit;
```

In this case,
1) we read a value,
2) check a condition over the value, and
3) write depending on the condition

## Another Issue: Write Skew

- A hospital uses a database to keep track of doctors and who are *on call* (jourläkare)
- For each shift, there needs to be at least one doctor on call

| name | on_call | shift_id |
|---|---|---|
| Alice | true | 1234 |
| Bob | true | 1234 |
| Carol | false | 1234 |

# Write Skew

Time

**Alice**

```
start transaction;
on_call = (
  select count(*) from doctors
    where on_call = true
    and shift_id = 1234
)
if (on_call >= 2) {
  update doctors
    set on_call = false
    where name = 'Alice'
    and shift_id = 1234
}
commit;
```

| name | on_call |
|---|---|
| Alice | true |
| Bob | true |
| Carol | false |

**Bob**

```
start transaction;
on_call = (
  select count(*) from doctors
    where on_call = true
    and shift_id = 1234
)




if (on_call >= 2) {
  update doctors
    set on_call = false
    where name = 'Bob'
    and shift_id = 1234
}
commit;
```

| name | on_call |
|---|---|
| Alice | false |
| Bob | false |
| Carol | false |

Note that both transactions modify different rows!

## Write Skew

- The previous example is an example of **Write Skew**
  - It’s not a lost update since two different objects/rows are modified
- **Shape**
  1. Two transactions read the same rows
  2. They check some condition over the returned rows
  3. They change different rows, but all changes together invalidate the condition

# Solutions to Write Skew

- It might be possible to use constraints for certain cases
- Use the isolation level **Serializable**
- Use explicit locks:

  ```
  SELECT *
  FROM doctors
  WHERE on_call = true AND shift_id = 1234
  FOR UPDATE
  ```

  Note that COUNT(*) has been replaced with * to lock the returned rows. (*If the SELECT statement checks for the absence of rows, then you cannot attach locks.*)
- This effect, when one write query affects the result of a search query in another transaction, is called *phantom*.

## Isolation Level: Serializable

- The isolation level **Serializable** protects against all shown issues, but with a performance cost.
- In some situations it might be better to use Serializable than to use locks, because concurrency bugs can be very tricky!
- Serializable has the same effect as running transactions in serial
- Implementation techniques:
  - Actual serial execution
  - Two-Phase Locking (2PL)
  - Serializable Snapshot Isolation (SSI)

## Transactions in JDBC

Transaction control in JDBC is performed by the Connection object.
When a connection is created, by default it is in auto-commit mode.

Turn off auto-commit mode (start a transaction) with:

```
conn.setAutoCommit(false);
// ... a series of statements treated as a transaction
conn.commit();
```

or

```
conn.rollback();
```

After a transaction, auto-commit mode must be turned on again.