

# Introduction to Data Management Transactions

Paul G. Allen School of Computer Science and Engineering  
University of Washington, Seattle


# Announcement

- Midterm is almost graded, to be released today
  - Scores appear highly correlated with attendance
- Final exam will be comprehensive:
  - Includes this material plus what we cover in 2<sup>nd</sup> half
- HW4 dues on Friday

# Terminology

Two types of query workloads:

- Online Analytical Processing (OLAP)
  - SELECT-FROM-WHERE are complex
  - No INSERT/UPDATE/DELETE, or very few
  - For data visualization (eg Tableau), or interactive SQL
- Online Transaction Processing (OLTP):
  - Lots of INSERT/UPDATE/DELETE
  - SELECT-FROM-WHERE are very simple
  - Used in Java/Python apps



We focused on these



Next few lectures

# Applications and Databases

Almost every app uses some database

- General purpose language (Java, Python)
- App issues SQL commands to RDBMS
- Usually, multiple apps (users) access same DB

# Simple Banking App in Python

- Manage user accounts:
  - Names
  - Balances
  - ...
  
- Allow users to:
  - Inquire balance
  - Deposit cash/check
  - Withdraw cash
  - Transfer money

# Simple Banking App in Python

SQL

```
CREATE TABLE Acc (  
    Usr TEXT PRIMARY KEY,  
    Balance INT);
```

Acc

Usr	Balance
Alice	300
Bob	600
Carol	400

# Simple Banking App in Python

## SQL

```
CREATE TABLE Acc (  
    Usr TEXT PRIMARY KEY,  
    Balance INT);
```

## Acc

Usr	Balance
Alice	300
Bob	600
Carol	400

## Python\*

```
import sqlite3  
con = sqlite3.connect("/Users/suciu/temp/bank.db",  
                      autocommit=True)  
  
cur = con.cursor()  
res = cur.execute("SELECT * FROM acc")  
answ = res.fetchall()  
print("The answer is: ", answ)
```

\* Documentation here <https://docs.python.org/3/library/sqlite3.html>

# Simple Banking App in Python

## SQL

```
CREATE TABLE Acc (  
    Usr TEXT PRIMARY KEY,  
    Balance INT);
```

## Acc

Usr	Balance
Alice	300
Bob	600
Carol	400

## Python\*

```
import sqlite3  
con = sqlite3.connect("/Users/suciu/temp/bank.db",  
                      autocommit=True)  
  
cur = con.cursor()  
res = cur.execute("SELECT * FROM acc")  
answ = res.fetchall()  
print("The answer is: ", answ)
```

SQL query  
sent to DBMS

\* Documentation here <https://docs.python.org/3/library/sqlite3.html>



DEMO:

lec16\_txn\_demo\_create\_table.sql

lec16\_txn\_demo\_simple\_1.py

# Terminology: Client/Server

## ■ Client:

- The program running the application
- In our example: a python program running on laptop
- In general: a big program on laptop or in the cloud

## ■ Server:

- The database management system
- In our example it is Sqlite on laptop
- In general: any RDBMS, on remote server or in cloud

# Parameterized Query

Give every user a 4% interest

```
res = cur.execute("SELECT * FROM acc")
answ = res.fetchall()
for row in answ:
    usr = row[0]
    bal = row[1]
    b = int(bal)
    i = b*0.04
    cur.execute("UPDATE acc
                SET balance=?
                WHERE usr=?",
                [b+i, usr])
```

# Parameterized Query

Give every user a 4% interest

Read data

```
res = cur.execute("SELECT * FROM acc")
answ = res.fetchall()
for row in answ:
    usr = row[0]
    bal = row[1]
    b = int(bal)
    i = b*0.04
    cur.execute("UPDATE acc
                SET balance=?
                WHERE usr=?",
                [b+i, usr])
```

# Parameterized Query

Give every user a 4% interest

```
res = cur.execute("SELECT * FROM acc")
answ = res.fetchall()
for row in answ:
    usr = row[0]
    bal = row[1]
    b = int(bal)
    i = b*0.04
    cur.execute("UPDATE acc
                SET balance=?
                WHERE usr=?",
                [b+i, usr])
```



Parameterized query

DEMO:

lec16\_txn\_demo\_simple\_2.py

# Simple Banking App in Python

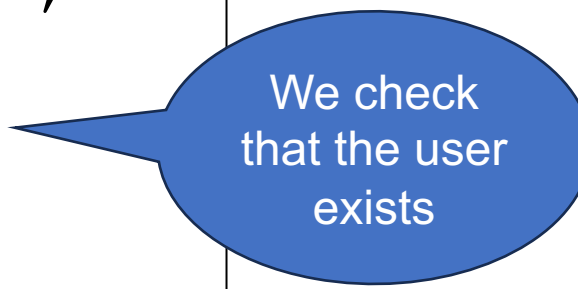
Our application should:

- Read a username
- Repeat:
  - Read a command
  - Execute that command
- The command can be:
  - Check the balance
  - Deposit money
  - Withdraw money
  - Transfer between accounts

# Simple Banking App in Python

Read a username, check if exists:

```
usr = input("Enter the user name: ")  
  
res = cur.execute("SELECT *  
                  FROM acc  
                  WHERE usr=?",  
                  [usr])  
  
if res.fetchone() is None:  
    print("Wrong user. Exit")  
    exit()
```



We check  
that the user  
exists



# Simple Banking App in Python

A simple loop for executing commants:

```
while True:
    cmd = input()
    if cmd == "b":    ... check balance
    elif cmd == "d":  ... deposit
    elif cmd == "w":  ... withdraw
    elif cmd == "t":  ... transfer
    elif cmd == "q":  exit()
```

# Simple Banking App in Python

## Check balance

```
res = cur.execute("SELECT balance
                  FROM acc
                  WHERE usr=?",
                  [usr])
row = res.fetchone()
b = row[0]
print("Balance is", b)
```

Fetch one  
row/tuple  
from output

First element  
of the tuple

# Simple Banking App in Python

## Deposit

```
... Read the balance b as before
amount = input() # amount to be deposited
a = int(amount)
b1 = b+a          # the new balance
cur.execute("UPDATE acc
            SET balance = ?
            WHERE usr=?",
            [b1,usr])
```

# Simple Banking App in Python


## Withdraw

```
... Read the balance b as before
amount = input() # amount to be withdrawn
a = int(amount)
#
# THE BANK DISPENSES MONEY HERE!
#
b1 = b-a          # the new balance
cur.execute("UPDATE acc
            SET balance = ?
            WHERE usr=?",
            [b1,usr])
```

# Simple Banking App in Python

## Withdraw

```
... Read the balance b as before
amount = input() # amount to be withdrawn
a = int(amount)
#
# THE BANK DISPENSES MONEY HERE!
#
b1 = b-a          # the new balance
cur.execute("UPDATE acc
            SET balance = ?
            WHERE usr=?",
            [b1,usr])
```



We need to check  
if there is enough  
money!

# Simple Banking App in Python

## Withdraw

```
... Read the balance b as before
amount = input() # amount to be withdrawn
a = int(amount)
if a>b:          # error: overdraft!
    exit()
#
# THE BANK DISPENSES MONEY HERE!
#
b1 = b-a          # the new balance
cur.execute("UPDATE acc
            SET balance = ?
            WHERE user=?",
            [b1,usr])
```

Better now

# Simple Banking App in Python

## Transfer

```
... Read the balance b as before
amount = input() # amount to be transferred
a = int(amount)
if a>b:          # error: overdraft!
    exit()
usr = input()   # to whom to transfer
... Read the balance bt of usr
b1 = b-a
bt1 = bt+a
cur.execute("UPDATE acc
            SET balance = ?
            WHERE user=?",
            [b1,usr])
cur.execute("UPDATE acc
            SET balance = ?
            WHERE user=?",
            [bt1,usr])
```

DEMO:

lec16\_txn\_demo.py  
single user



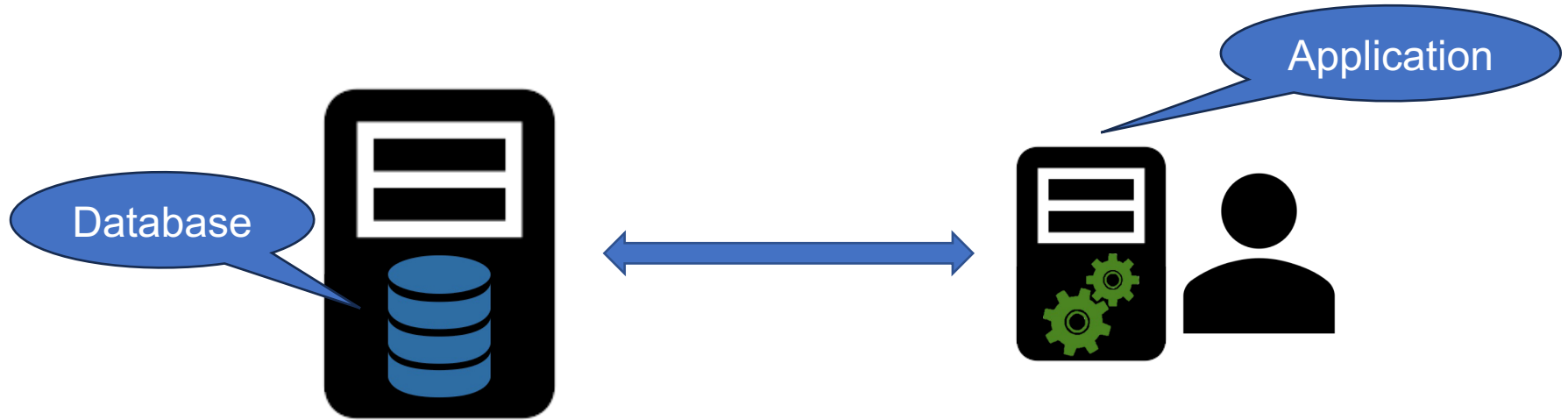
# Discussion so Far

- The users Alice, Bob, ... don't need to know SQL, but interact with the app;
- The app usually has a nice User Interface (UI)
- The database is persistent: it retains the data for a long period of time

# Concurrency

# Single-Server

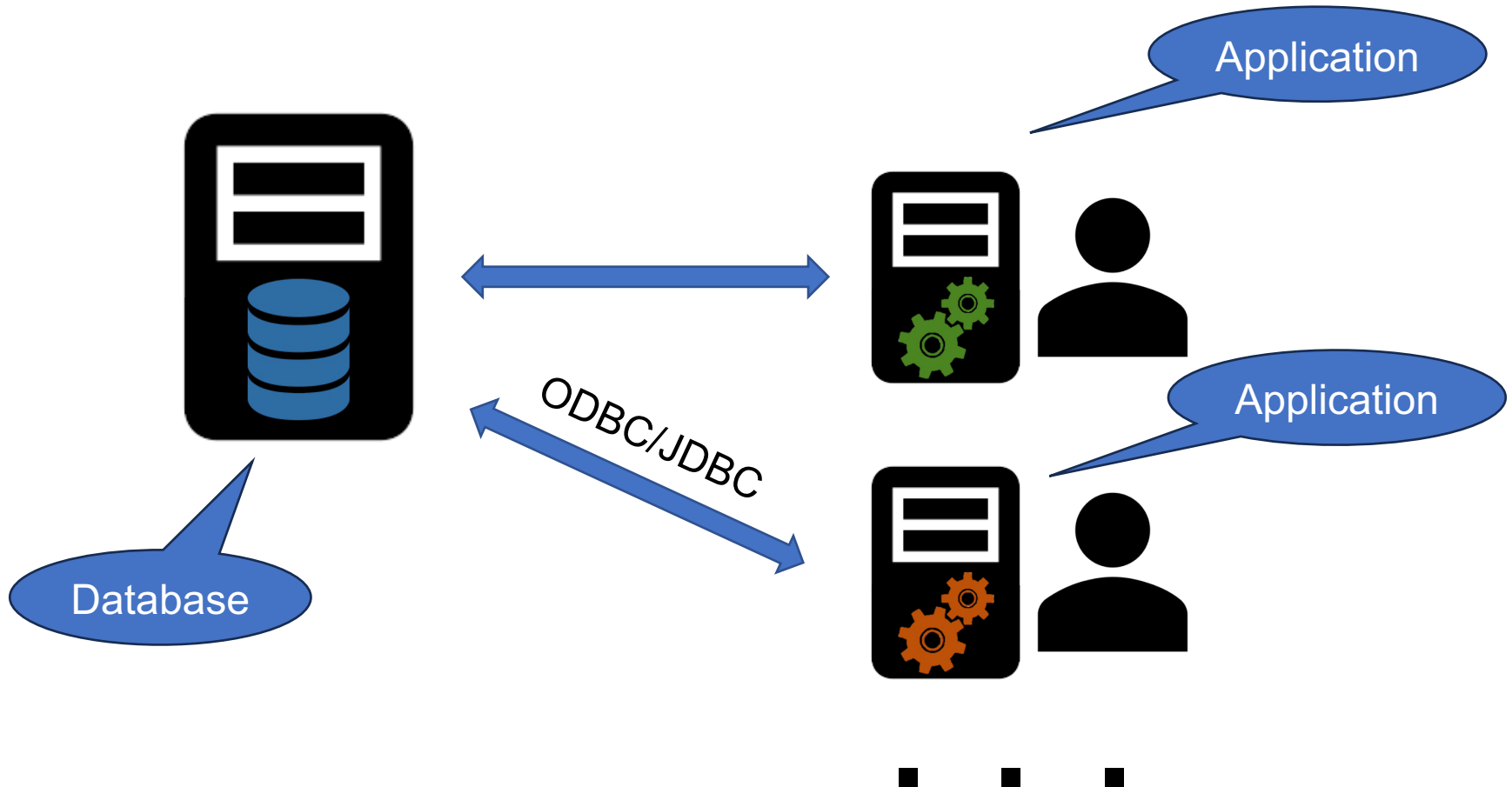
- The database is accessed by a single user:



- RDBMS on same laptop, or a server, or the cloud

# Client-Server or Two-Tier Architecture

- Multiple users access the database concurrently



DEMO:

lec16\_txn\_demo.py  
multiple users

lec16\_txn\_demo\_txn\_no.sql

# What We Have Seen

How Alice and Bob colluded to steal \$100 (simplified, using only SQL)  
Current balance of Alice is \$100:

```
-- Alice withdraws $100
b = SELECT balance
  FROM acc
  WHERE user = 'Alice';
-- Is b >= 100?  Yes:
-- Dispense money

UPDATE acc SET balance=b-100
WHERE user = 'Alice'
```



```
-- Bob impersonates Alice
-- and also withdraws $100
b = SELECT balance
  FROM acc
  WHERE user = 'Alice';
-- Is b >= 100?  Yes:
-- Dispense money
UPDATE acc SET balance=b-100
WHERE user = 'Alice'
```

# Discussion

- Users Alice, Bob, ... can access the same database concurrently
- This may lead to the database being inconsistent, which is a big problem

# Consistency



# Database Consistency

- Consistency: a property that should always hold
  - Every account balance is  $\geq 0$
  - The sum of all balances is constant, or changes exactly by the amount deposited/withdrawn
- If we write the application correctly, we expect the database to remain consistent
- But (without transactions!) things can go wrong during concurrency. Next.

# Conflicts Between Concurrent Operations

# Common Concurrency Conflicts

- Lost Update
- Dirty/Inconsistent Read
- Unrepeatable Read
- Phantom Read

These have popular names, but all sorts of other conflicts can happen. Let's see these.

# Dirty/Inconsistent Read

A **inconsistent read** happens when data is read "during" a write

- **Dirty/Inconsistent Read**
- Unrepeatable Read
- Lost Update

*Manager wants to  
balance project budgets*

*CEO wants to check  
company balance*

time



# Dirty/Inconsistent Read

A **inconsistent read** happens when data is read "during" a write

- **Dirty/Inconsistent Read**
- Unrepeatable Read
- Lost Update

*Manager wants to  
balance project budgets*

-\$10mil from project A

+\$7mil to project B

+\$3mil to project C

*CEO wants to check  
company balance*

SELECT SUM(money) ...

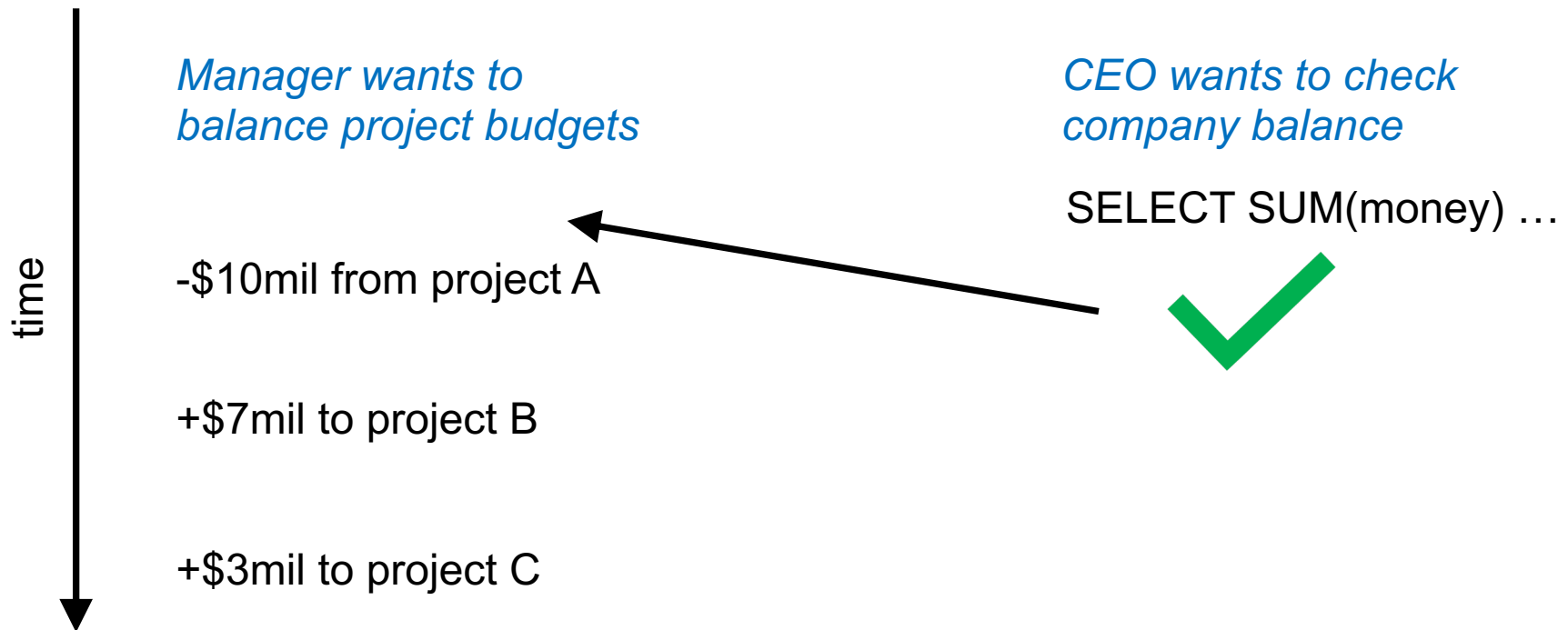
time



# Dirty/Inconsistent Read

A **inconsistent read** happens when data is read "during" a write

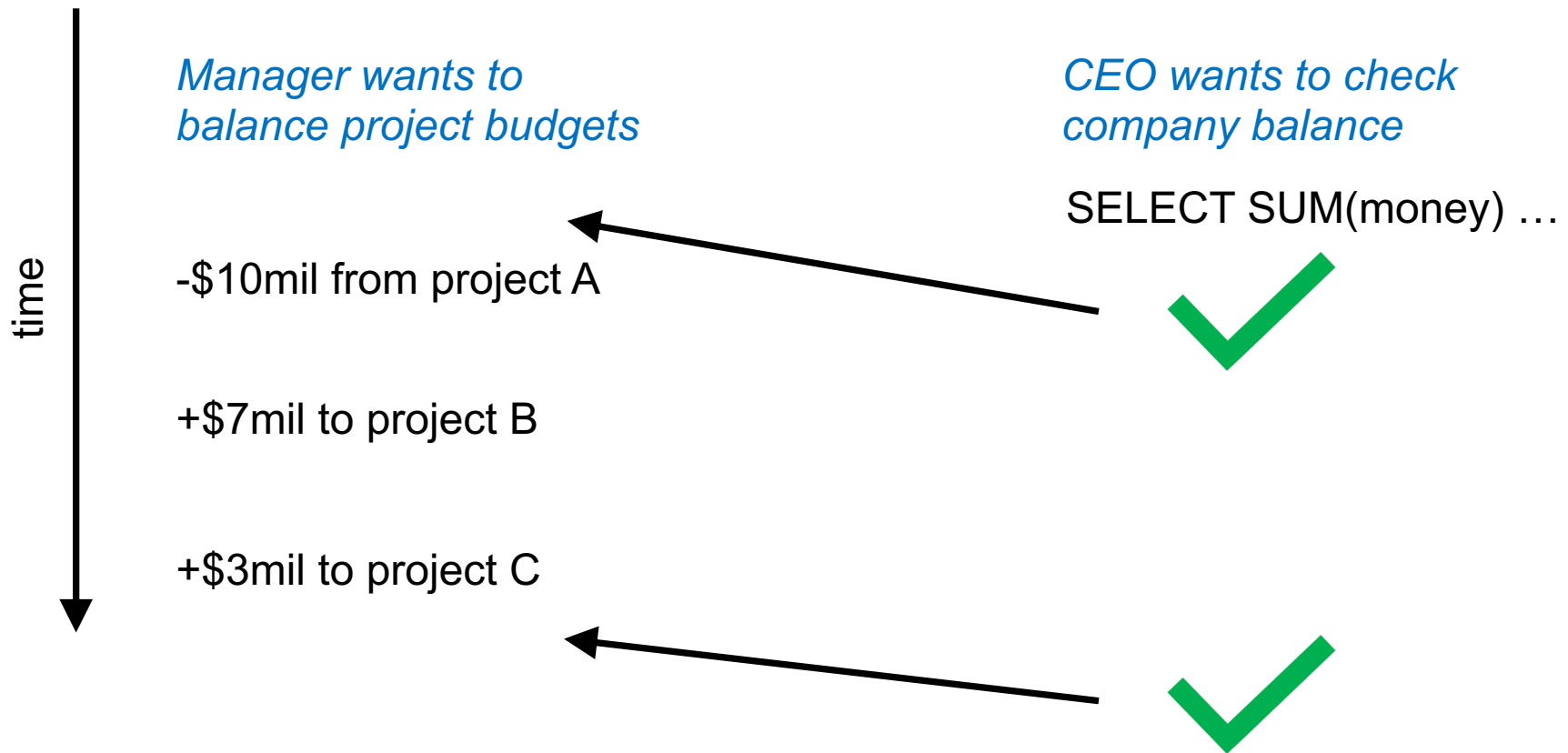
- **Dirty/Inconsistent Read**
- Unrepeatable Read
- Lost Update



# Dirty/Inconsistent Read

A **inconsistent read** happens when data is read "during" a write

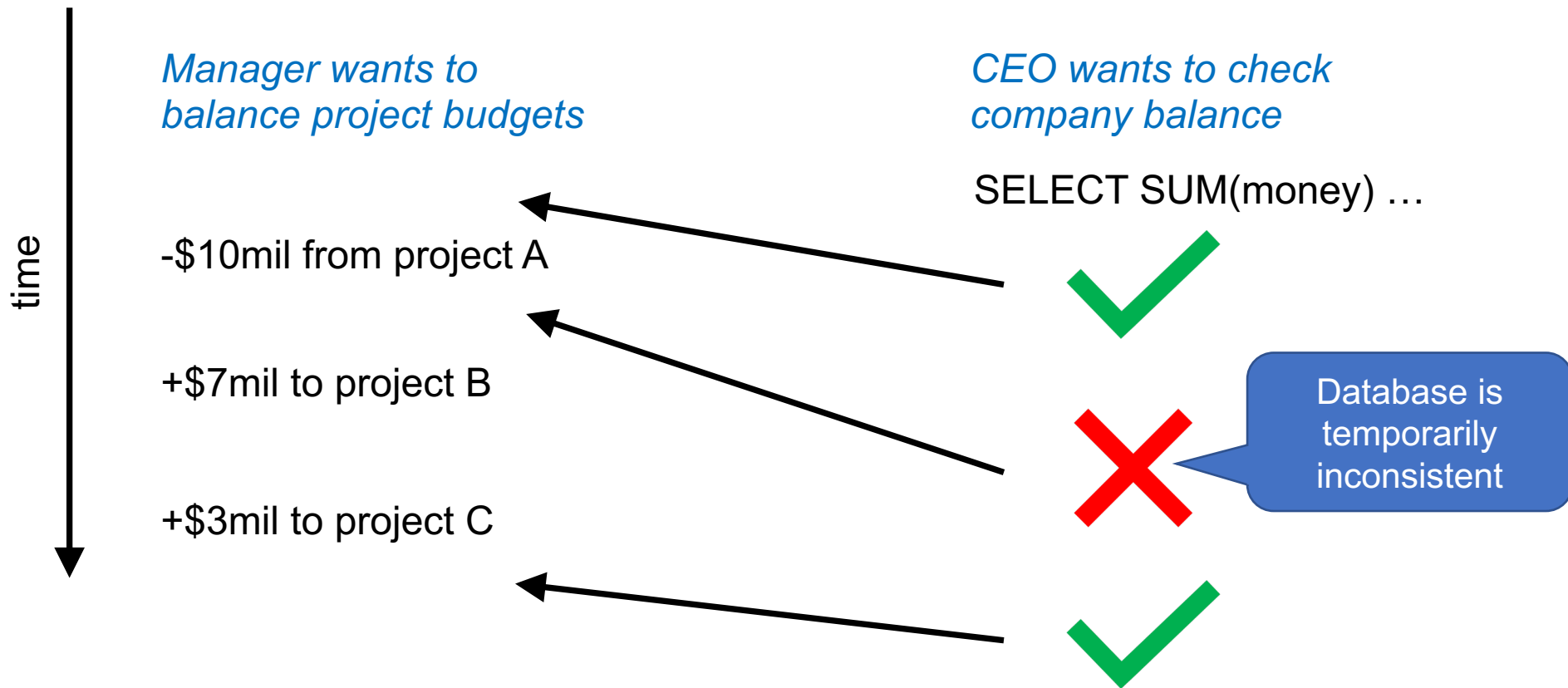
- **Dirty/Inconsistent Read**
- Unrepeatable Read
- Lost Update



# Dirty/Inconsistent Read

A **inconsistent read** happens when data is read "during" a write

- **Dirty/Inconsistent Read**
- Unrepeatable Read
- Lost Update





# Unrepeatable Read

An **unrepeatable read** happens when data read twice differs

- Dirty/Inconsistent Read
- **Unrepeatable Read**
- Lost Update

*Accountant wants to check company assets*

SELECT inventory  
FROM Products  
WHERE pid = 1

SELECT inventory\*price  
FROM Products  
WHERE pid = 1

*Warehouse updates inventory levels*

UPDATE Products  
SET inventory = 0  
WHERE pid = 1

Might get a value that doesn't correspond to previous read!

# Aside: Phantom Read

- Dirty/Inconsistent Read
- Unrepeatable Read
- Lost Update

A **phantom read** happens when a record is inserted/delete during reads

*Accountant wants to check company assets*

SELECT \*  
FROM products  
WHERE price < 10.00

*Warehouse receives new products*

INSERT INTO Products  
VALUES ('nuts', 10, 8.99)

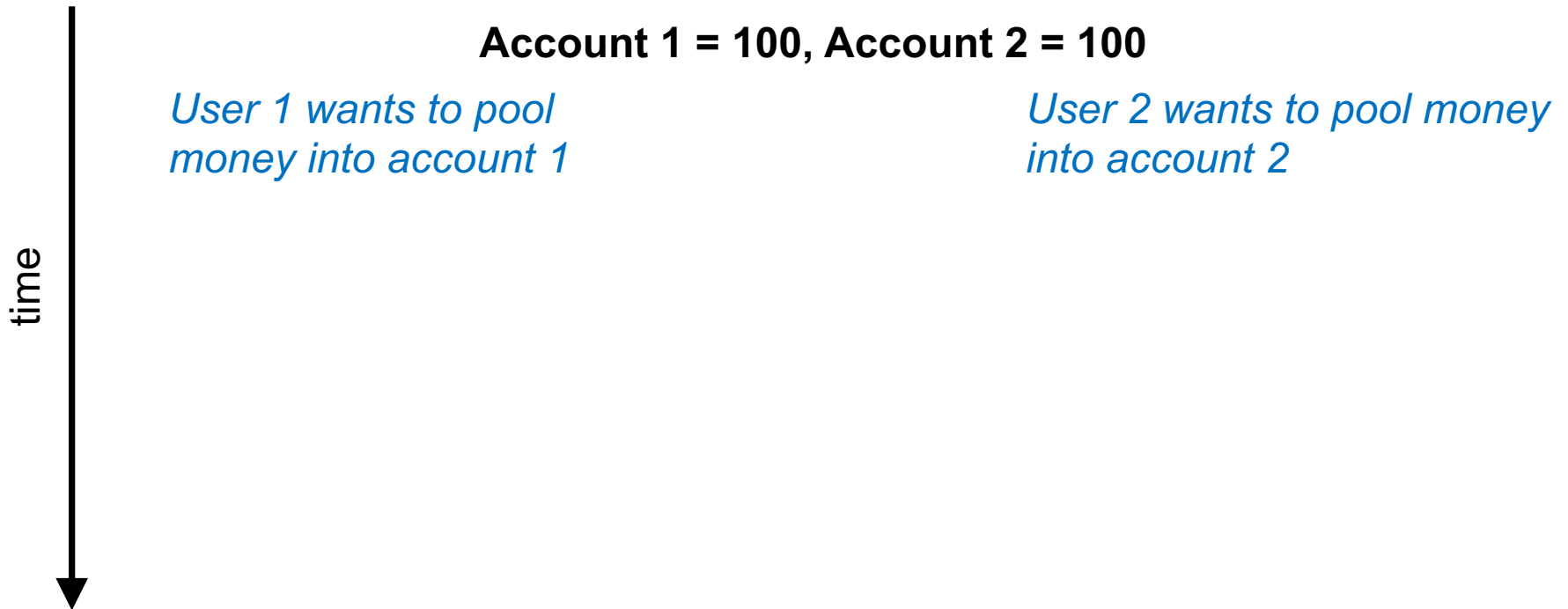
SELECT \*  
FROM products  
WHERE price < 20.00

Returns a “new” row that should have been in the last read!

# Lost Update

A **lost update** happens when a write "disappears"

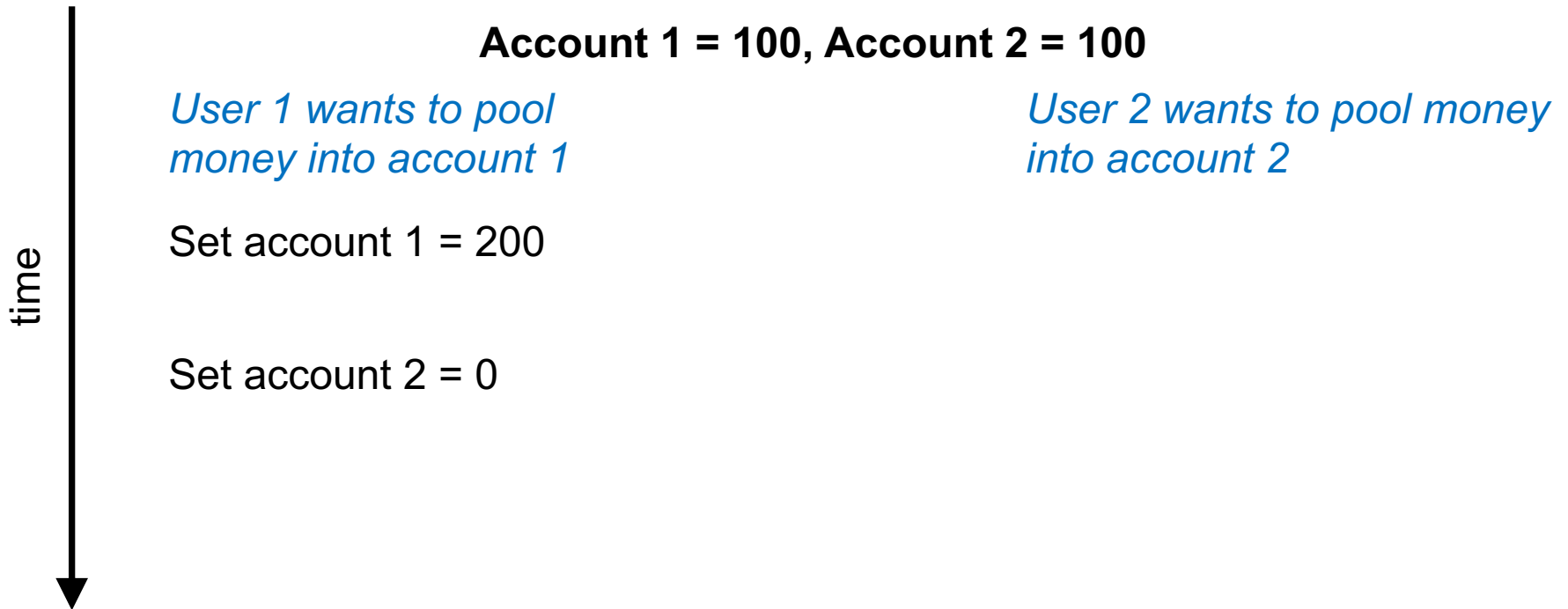
- Dirty/Inconsistent Read
- Unrepeatable Read
- **Lost Update**



# Lost Update

A **lost update** happens when a write "disappears"

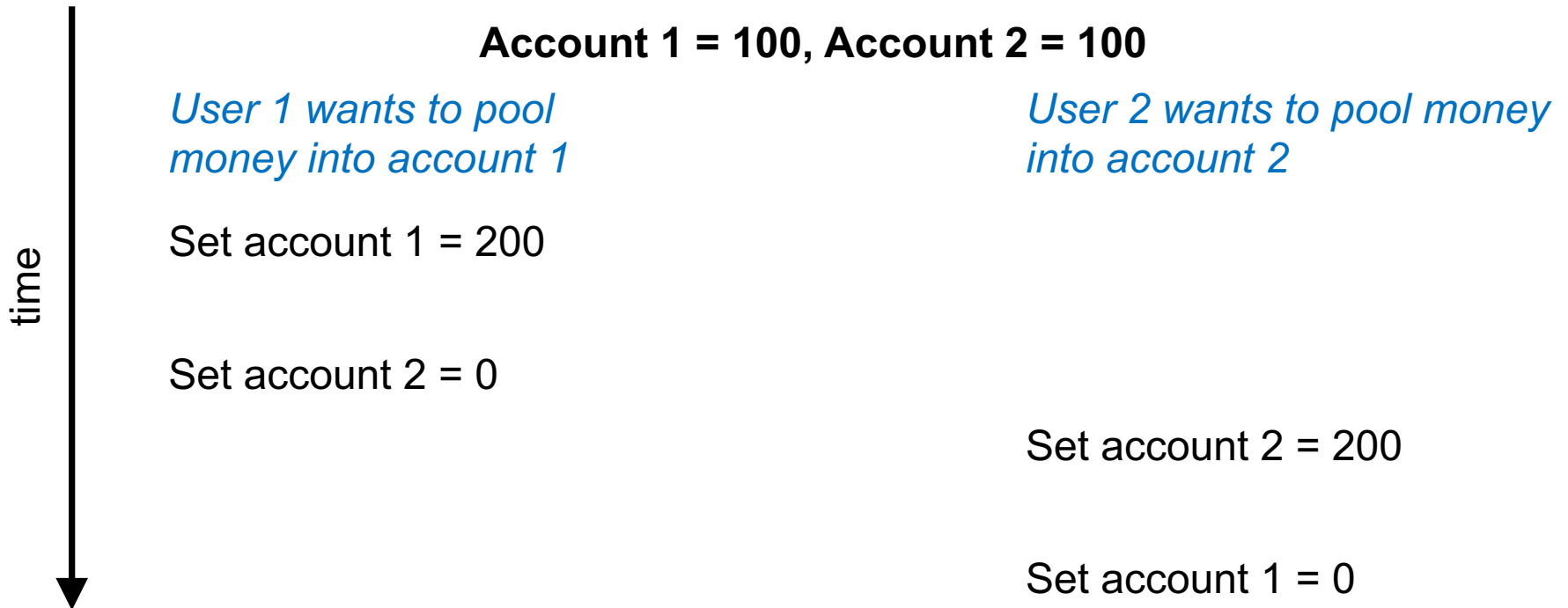
- Dirty/Inconsistent Read
- Unrepeatable Read
- **Lost Update**



# Lost Update

A **lost update** happens when a write "disappears"

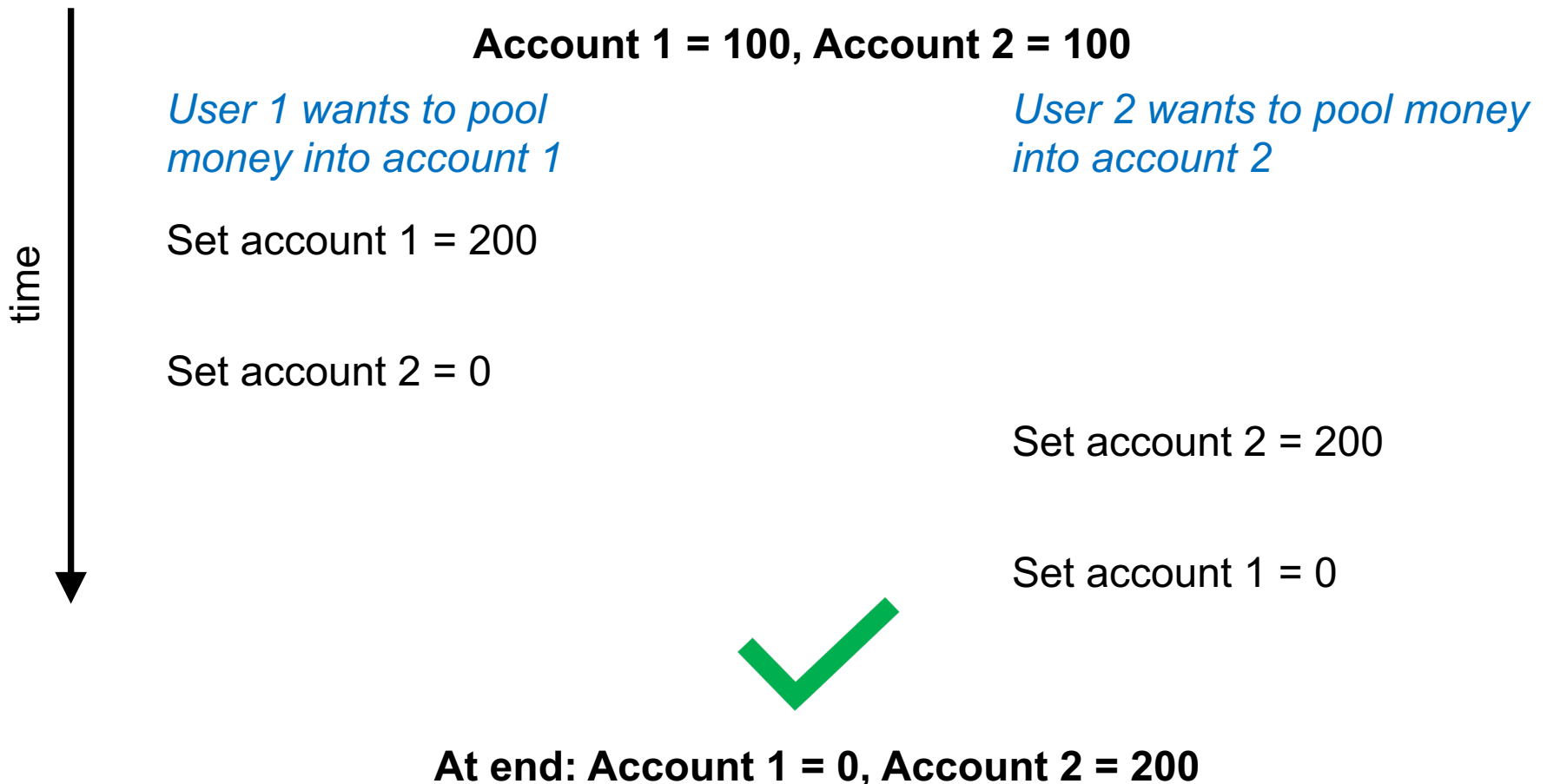
- Dirty/Inconsistent Read
- Unrepeatable Read
- **Lost Update**



# Lost Update

A **lost update** happens when a write "disappears"

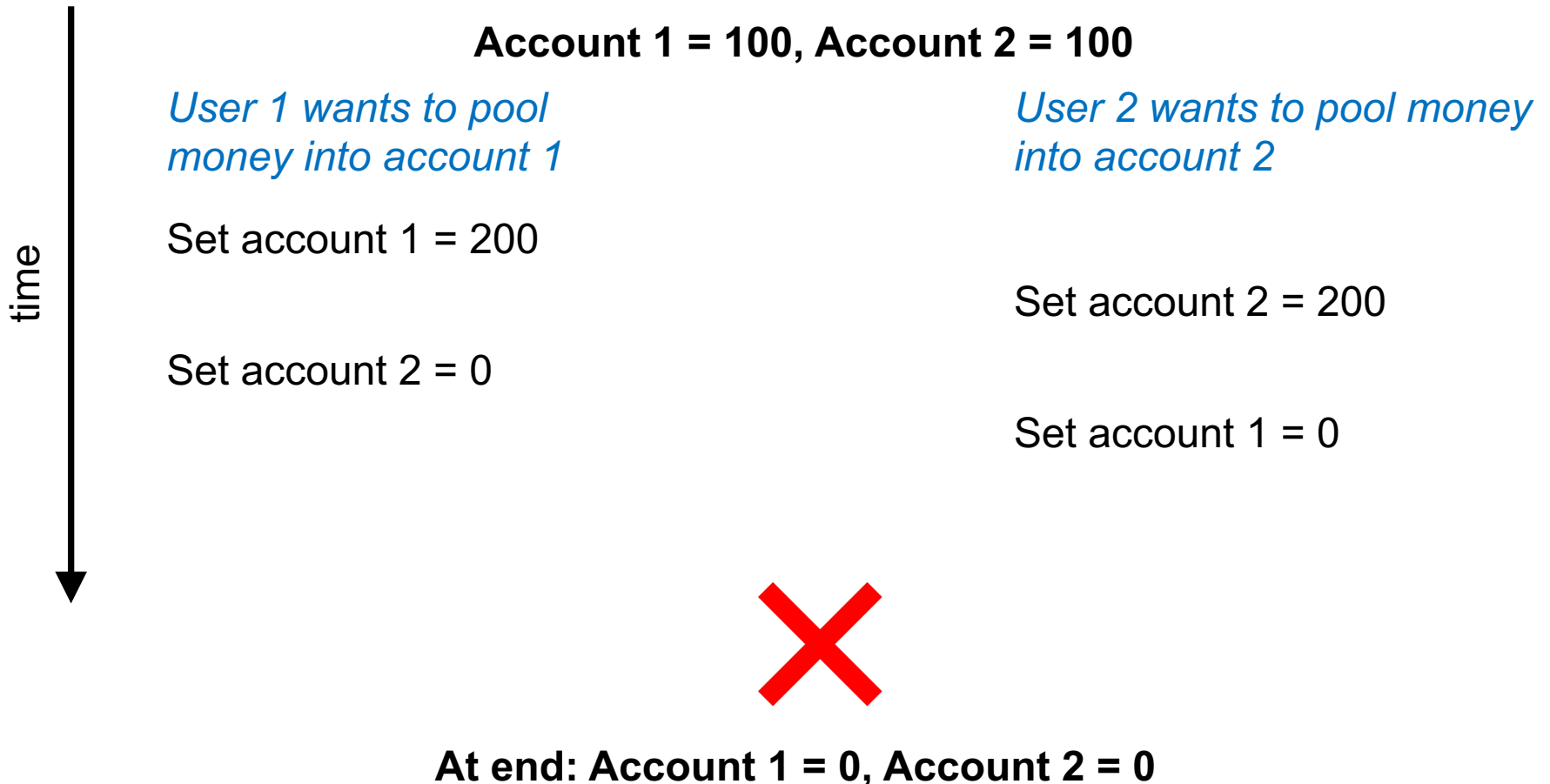
- Dirty/Inconsistent Read
- Unrepeatable Read
- **Lost Update**



# Lost Update

A **lost update** happens when a write "disappears"

- Dirty/Inconsistent Read
- Unrepeatable Read
- **Lost Update**



# Transactions



# Transactions

- A transaction is a set of read and writes to the database that execute all or nothing

**BEGIN TRANSACTION**

...SQL Statements

**COMMIT**

Entire txn is executed

**BEGIN TRANSACTION**

...SQL Statements

**ROLLBACK**

No part of txn is executed

# Transactions

- Prevent all concurrency control conflicts
- Easy to use in app: group statements in txns
- Let's see how they work

DEMO:

lec16\_txn\_demo\_txn\_yes.sql