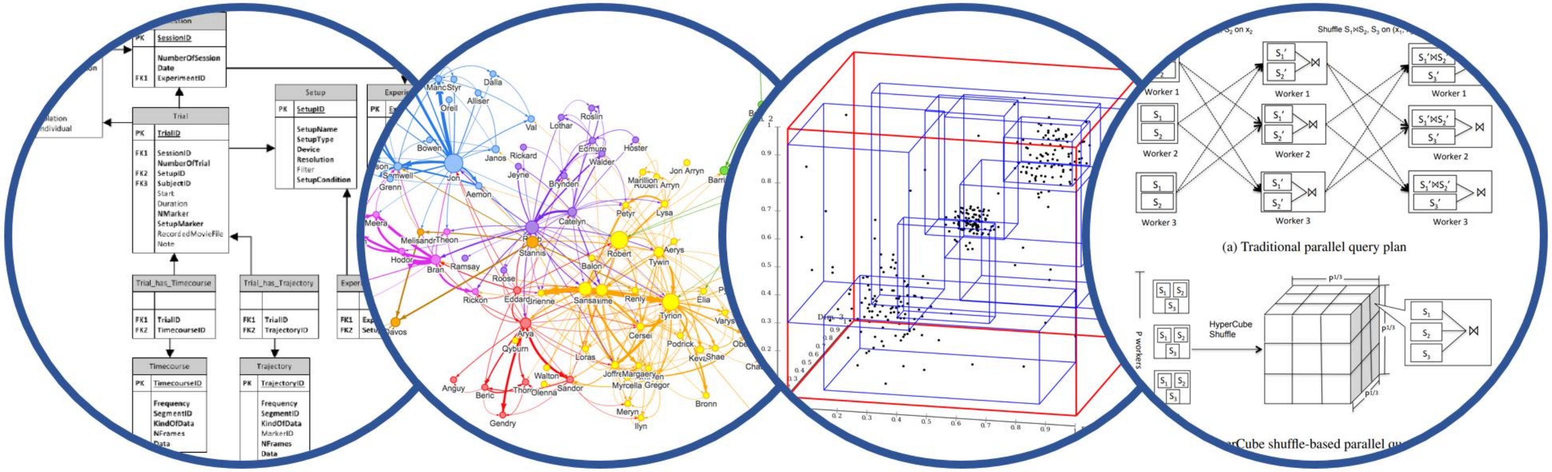


Introduction to Data Management

Transactions

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

Terminology

Two types of query workloads:

- Online Analytical Processing (OLAP)
  - SELECT-FROM-WHERE are complex
  - No INSERT/UPDATE/DELTE, or very few
  - For data visualization (eg Tableau), or interactive SQL

We focused on these

- Online Transaction Processing (OLTP):
  - Lots of INSERT/UPDATE/DELETE
  - SELECT-FROM-WHERE are very simple
  - Used in Java/Python apps

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(”/path/to/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("/path/to/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:

txn_demo_create_table.sql

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 = float(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 = float(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 = float(bal)
    i = b*0.04
    cur.execute("UPDATE acc
                 SET balance=?
                 WHERE usr=?",
                 [b+i, usr])
```

Parameterized query

DEMO:

txn_demo_simple_2.py

Simple Banking App in Python

Read a username

Repeat:

- Read a command
- Execute that command
  - 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()
usrt = input()     # to whom to transfer
... Read the balance bt of usrt
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,usrt])
```

DEMO:

txn_demo.py

single user

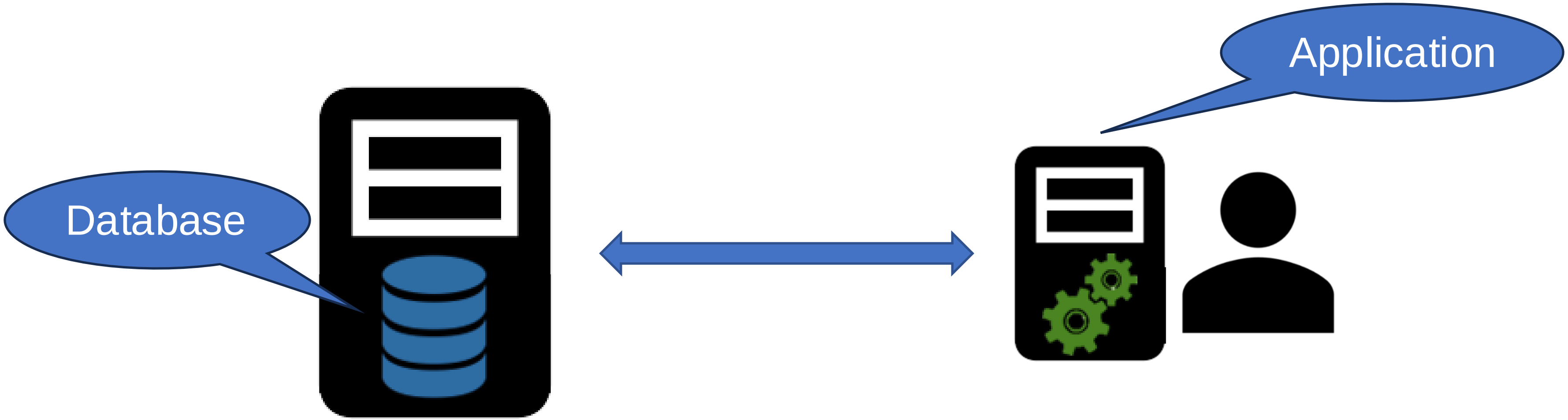
Discussion so Far

- 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

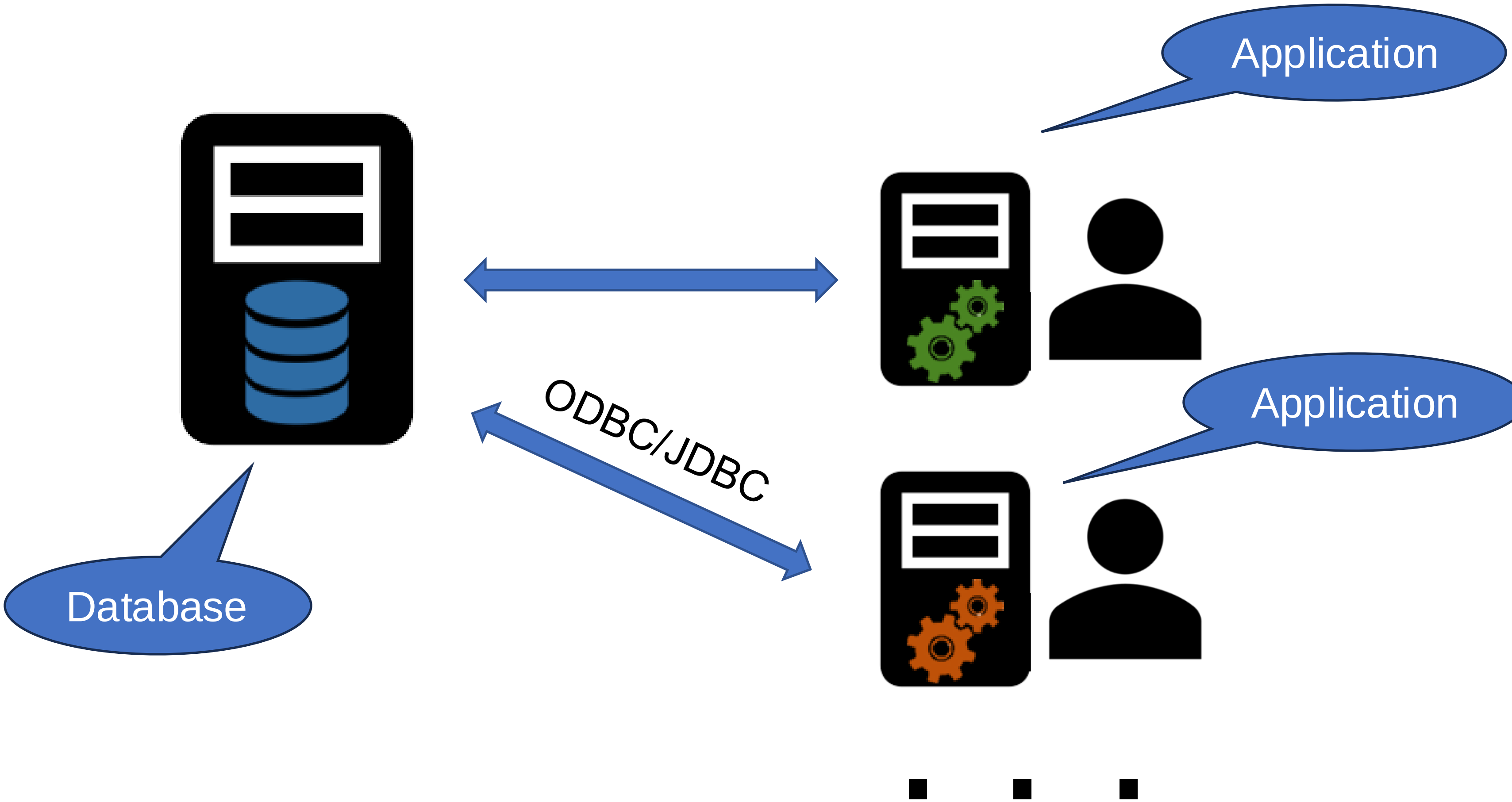
- The database is accessed by a single user:



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

Client-Server or Two-Tier Architecture

- Multiple users access the database concurrently



DEMO:

txn_demo.py

multiple users

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'
```

time

```
-- 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 ≥ 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

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

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

Dirty/Inconsistent Read

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

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

time

Manager wants to balance project budgets

CEO wants to check company balance

Dirty/Inconsistent Read

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

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

time

Manager wants to balance project budgets

CEO wants to check company balance

SELECT SUM(money) …

-$10mil from project A

+$7mil to project B

+$3mil to project C

Dirty/Inconsistent Read

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

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

time

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) …

Dirty/Inconsistent Read

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

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

time

Manager wants to balance project budgets

CEO wants to check company balance

SELECT SUM(money) …

-$10mil from project A

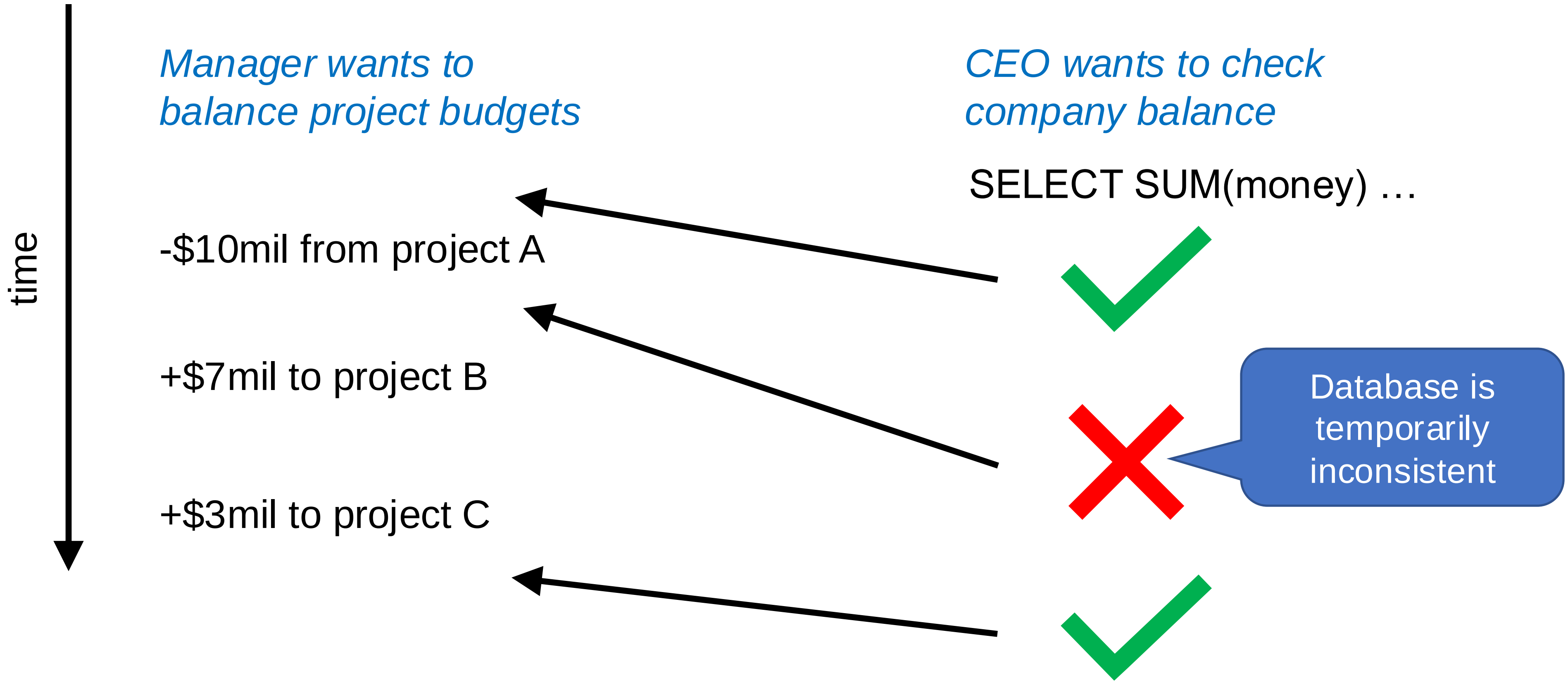
+$7mil to project B

+$3mil to project C

Dirty/Inconsistent Read

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

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



Unrepeatable Read

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

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

time

Accountant wants to check company assets

Warehouse updates inventory levels

```
SELECT inventory
FROM Products
WHERE pid = 1
```

```
UPDATE Products
SET inventory = 0
WHERE pid = 1
```

```
SELECT inventory*price
FROM Products
WHERE pid = 1
```

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

Phantom Read

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

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

time

Accountant wants to check company assets

Warehouse receives new products

```
SELECT *
FROM products
WHERE price < 20.00
```

```
INSERT INTO Products
VALUES (‘nuts’, 10, 8.99)
```

```
SELECT *
FROM products
WHERE price < 10.00
```

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
- Phantom Read
- **Lost Update**

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1

User 2 wants to pool money into account 2

Lost Update

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

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

time

Account 1 = 100, Account 2 = 100

User 1	User 2
User 1 wants to pool money into account 1	*User 2 wants to pool money into account 2*
Set account 1 = 200	
Set account 2 = 0	

Lost Update

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

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

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1	*User 2 wants to pool money into account 2*
Set account 1 = 200	
Set account 2 = 0	
	Set account 2 = 200
	Set account 1 = 0

Lost Update

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

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

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1	*User 2 wants to pool money into account 2*
Set account 1 = 200	
Set account 2 = 0	
	Set account 2 = 200
	Set account 1 = 0

At end: Account 1 = 0, Account 2 = 200

Lost Update

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

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

time

Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1	*User 2 wants to pool money into account 2*
Set account 1 = 200	
	Set account 2 = 200
Set account 2 = 0	
	Set account 1 = 0

At end: Account 1 = 0, Account 2 = 0

Transactions

Transactions

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

```
BEGIN TRANSACTION
   ...SQL Statements
COMMIT
```

```
BEGIN TRANSACTION
   ...SQL Statements
ROLLBACK
```

Entire txn is executed

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:
txn_demo_txn_yes.sql

Transactions

- Prevent all concurrency control conflicts
- Easy to use in app: group statements in txns
- What property does a TXN satisfy?
  - Informally: “TXNs have ACID properties”
  - Formally: “execution of TXNs must be serializable”

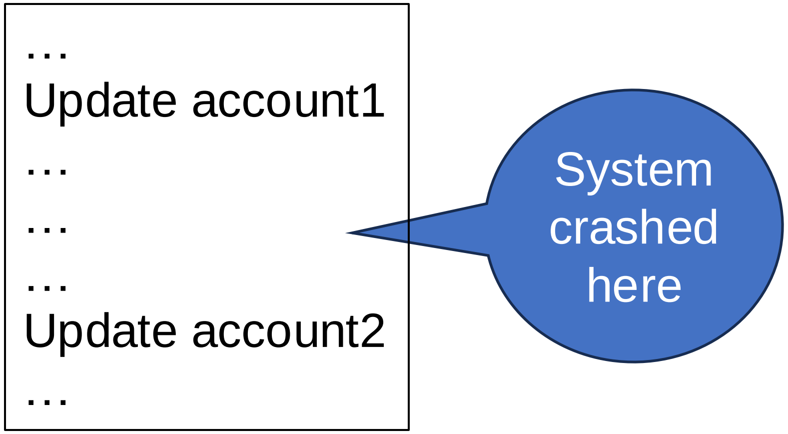
ACID

Transactions are ACID

- Atomic
- Consistent
- Isolated
- Durable

Atomic

- A set of operations is atomic if either all its operations happen, or none happens



Recovery manager (not discussed in this class)

Consistent

Assume TXN is “correct” (this is app specific)

- If TXN starts with the DB in a consistent state, it must end leaving the DB in a consistent state

It is a consequence of Atomicity and Isolation

Isolated

- The effect of the transaction on the database is as if it were running alone on the database

TXN1:
…
Update account1
…
…
…
Update account2
…

TXN2:
…
Update account1
…
…
…
Update account2
…

Interleaved actions should not interfere

Concurrency Control Manager

Durable

- Data should be stored persistently on disk, always in a consistent state

Discussion

- ACID properties: popular job interview question

- "A" and "I" matter
  - Atomicity: recover from crashes
  - Isolation: concurrency control

444

344 and 444

- ACID is informal.

Will discuss the formal property next

Serializability

Problem Definition

- The RDBMs runs several TXNs: T1, T2, T3, …

- It could run T1 to completion before starting T2,
  then run T2 to completion before starting T3,
  then run T3…
  …

Problem Definition

- The RDBMs runs several TXNs: T1, T2, T3, …

- It could run T1 to completion before starting T2,
  then run T2 to completion before starting T3,
  then run T3…
  …
  But this has poor performance

Why?

Problem Definition

- The RDBMs runs several TXNs: T1, T2, T3, …

- It could run T1 to completion before starting T2,
  then run T2 to completion before starting T3,
  then run T3…
  …
  But this has poor performance   Why?

- Instead: interleave commands from multiple TXNs

When is the interleaving ”safe”?

Simplified Data Model for TXN

Element is usually
a record, or a disk block

- Database = a set of “elements”
- TXN = a sequence of Reads/Writes of elements

Example

T1
READ(A, t)
t := t+100
WRITE(A, t)
READ(B, t)
t := t+100
WRITE(B,t)

Example

T1
READ(A, t)
t := t+100
WRITE(A, t)
READ(B, t)
t := t+100
WRITE(B,t)

A,B are elements in the DB

Example

T1
READ(A, t)
t := t+100
WRITE(A, t)
READ(B, t)
t := t+100
WRITE(B,t)

A,B are elements in the DB

t is a local variable in the app

Example

T1
READ(A, t)
t := t+100
WRITE(A, t)
READ(B, t)
t := t+100
WRITE(B,t)

A,B are elements in the DB

t is a local variable in the app

T2
READ(A, s)
s := s*2
WRITE(A,s)
READ(B,s)
s := s*2
WRITE(B,s)

Definitions

- An interleaving of READ/WRITEs from different TXNs is called a schedule

Definitions

- An interleaving of READ/WRITEs from different TXNs is called a schedule

- **Definition**: a serial schedule is a schedule where all operations of transactions come before those of the next transaction

Definitions

- An interleaving of READ/WRITEs from different TXNs is called a schedule

- **Definition**: a serial schedule is a schedule where all operations of transactions come before those of the next transaction

- **Definition**: a serializable schedule is a schedule that is equivalent to a serial schedule

A Schedule

T1	T2
READ(A, t)	
	READ(A, s)
	s := s*2
t := t+100	
WRITE(A, t)	
	WRITE(A,s)
	READ(B,s)
	s := s*2
READ(B, t)	
	WRITE(B,s)
t := t+100	
WRITE(B,t)	

time

A Serial Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

time

A Serial Schedule

A = 2
B = 2

time

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

A Serial Schedule

A = 2
B = 2

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

A = 102
B = 102

time

A Serial Schedule

time	T1	T2
A = 2 B = 2	**READ**(A, t)	
	t := t+100	
	WRITE(A, t)	
	READ(B, t)	
	t := t+100	
	WRITE(B,t)	
A = 102 B = 102		**READ**(A, s)
		s := s*2
		WRITE(A,s)
		READ(B,s)
		s := s*2
A = 204 B = 204		**WRITE**(B,s)

The Other Serial Schedule

A = 2
B = 2

T1	T2
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

A = 4
B = 4

A = 104
B = 104

time

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

A = 2
B = 2

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

A = 2
B = 2

A = 102
B = 2

A Serializable Schedule

T1	T2	
		A = 2 B = 2
READ(A, t)		
t := t+100		
WRITE(A, t)		A = 102 B = 2
	READ(A, s)	
	s := s*2	
	WRITE(A,s)	A = 204 B = 2
READ(B, t)		
t := t+100		
WRITE(B,t)		
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	

A Serializable Schedule

T1	T2	
READ(A, t)		A = 2 B = 2
t := t+100		
WRITE(A, t)		A = 102 B = 2
	READ(A, s)	
	s := s*2	
	WRITE(A,s)	A = 204 B = 2
READ(B, t)		
t := t+100		
WRITE(B,t)		A = 204 B = 102
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	A = 204 B = 204

A Serializable Schedule

T1	T2	
		A = 2 B = 2
READ(A, t)		
t := t+100		
WRITE(A, t)		A = 102 B = 2
	READ(A, s)	
	s := s*2	
	WRITE(A,s)	A = 204 B = 2
READ(B, t)		
t := t+100		
WRITE(B,t)		A = 204 B = 102
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	A = 204 B = 204

This is NOT a serial schedule
It is a serializable schedule.

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

A = 2
B = 2

A = 102
B = 2

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

A = 2
B = 2

A = 102
B = 2

A = 204
B = 2

A Non-Serializable Schedule

T1	T2	
READ(A, t)		A = 2 B = 2
t := t+100		
WRITE(A, t)		A = 102 B = 2
	READ(A, s)	
	s := s*2	
	WRITE(A,s)	A = 204 B = 2
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	A = 204 B = 4
READ(B, t)		
t := t+100		
WRITE(B,t)		

A Non-Serializable Schedule

T1	T2	
		A = 2 B = 2
READ(A, t)		
t := t+100		
WRITE(A, t)		A = 102 B = 2
	READ(A, s)	
	s := s*2	
	WRITE(A,s)	A = 204 B = 2
	READ(B,s)	
	s := s*2	
	WRITE(B,s)	A = 204 B = 4
READ(B, t)		
t := t+100		
WRITE(B,t)		A = 204 B = 104

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A, s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

A = 2
B = 2

A = 102
B = 2

A = 204
B = 2

A = 204
B = 4

Should be impossible!

A = 204
B = 104

Discussion

- If the schedule is serial, then nothing can go wrong
- Same for a serializable schedule
- Concurrency Control Manager of the RDBMs must ensure that the schedule is serializable

How do we check that a schedule is serializable?

Conflict Serializability

Outline

We further simplify the model:

- A transaction is a sequence of reads and writes
- We ignore operations between reads and writes

Example

T1
READ(A, t)
t := t+100
WRITE(A, t)
READ(B, t)
t := t+100
WRITE(B,t)

→

T1
R(A)
W(A)
R(B)
W(B)

Also: $R_1(A), W_1(A), R_1(B), W_1(B)$

Example

- T1 then T2

$R_1(A), W_1(A), R_1(B), W_1(B), R_2(A), W_2(A), R_2(B), W_2(B)$

time

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Example

- T2 then T1

$R_2(A), W_2(A), R_2(B), W_2(B), R_1(A), W_1(A), R_1(B), W_1(B)$

T1	T2
	R(A)
	W(A)
	R(B)
	W(B)
R(A)	
W(A)	
R(B)	
W(B)	

Example

- Serializable to T1 then T2

$R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B)$

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Example

- Not serializable

$R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B)$

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Main Idea

- To check if a schedule is serializable, try swapping operations until it becomes serial:

… $R_i(A)$, $W_j(B)$, … → … $W_j(B)$, $R_i(A)$, …

- But we only swap if the new schedule is equivalent
- A pair is in conflict if it cannot be swapped

Conflicts

1. Any pair of ops of the same TXN are in conflict
2. $R_i(X)$, $W_j(X)$ forms a read-write conflict
3. $W_i(X)$, $R_j(X)$ forms a write-read conflict
4. $W_i(X)$, $W_j(X)$ forms a write-write conflict

Conflict Serializable Schedule

A schedule is <u>conflict serializable</u> if it can be **transformed** into a serial schedule by a series of swappings of adjacent **non-conflicting** actions

Conflict Serializable Schedule Example

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Conflict Serializable Schedule Example

T1	T2
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
	R(B)
	W(B)

Conflict Serializable Schedule Example

T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
	W(A)
W(B)	
	R(B)
	W(B)

Conflict Serializable Schedule Example

T1	T2
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
	R(B)
	W(B)

Conflict Serializable Schedule Example

T1	T2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Non Conflict Serializable Schedule Example

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Non Conflict Serializable Schedule Example

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
R(B)	
	W(B)
W(B)	

Conflict rule broken!

Serializable vs Conflict Serializable

Conflict serializability ignores what TXN does between the R's and the W's.
It assumes the worst / most complicated updates to the data

Serializable vs Conflict Serializable

Conflict serializability ignores what TXN does between the R's and the W's.
It assumes the worst / most complicated updates to the data

Not serializable nor conflict serializable

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

A←A+100 (T1: R(A), W(A))

A←A*2 (T2: R(A), W(A))

B←B*2 (T2: R(B), W(B))

B←B+100 (T1: R(B), W(B))

Serializable vs Conflict Serializable

Conflict serializability ignores what TXN does between the R's and the W's.
It assumes the worst / most complicated updates to the data

Serializable (because 100+2 = 2+100)
But not conflict serializable, because it assumes the worst

	T1	T2	
A←A+100	R(A)		
	W(A)		
		R(A)	A←A+2
		W(A)	
		R(B)	B←B+2
		W(B)	
B←B+100	R(B)		
	W(B)		

Discussion

- Most RDBMs enforce conflict-serializability
- Next: how to test for conflict-serializability

The Precedence Graph

Testing for Conflict Serializability

Fix a schedule

- **Definition**. The precedence graph has one node for every TXN in the schedule, and one edge for every pair of conflicting ops

- **Theorem**. The schedule is conflict-serializable iff the precedence graph has no cycles

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Nodes: 1 2 3

Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges:

$r_2(A)$ $r_1(B)$

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges: 1 2 3

$r_2(A)$ $r_1(B)$

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges: 1 2 3

$r_2(A)$ $r_1(B)$

No edge because
no conflict (A != B)

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges: ① ② ③

$r_2(A)$ $w_2(A)$

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges: 1 2 3

$r_2(A)$ $w_2(A)$

No edge because same txn (2)

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges:

1 2 3

$r_2(A)$ $r_3(A)$?

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges: ① ② ③

$r_2(A)$ $w_1(B)$?

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges: 1 2 3

$r_2(A)$ $w_3(A)$?

$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

Edges:

1 2 3

$r_2(A)$ $w_3(A)$

Edge! Conflict from T2 to T3

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

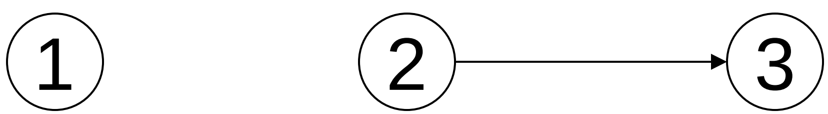
Edges: (1) (2) (3)

$r_2(A)$ $w_3(A)$

Edge! Conflict from T2 to T3

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

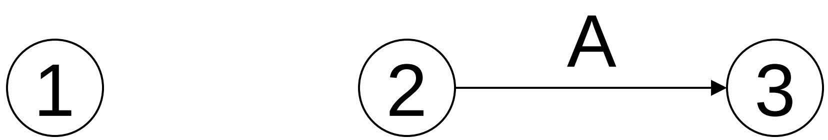
Edges:



$r_2(A)$ $r_2(B)$?

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges:

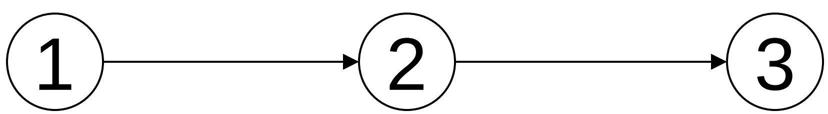


And so on until compared every pair of actions…

Example 1

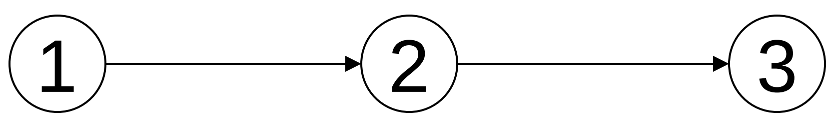
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges:



Repeating the same directed edge not necessary

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



This schedule is conflict-serializable

Example 2

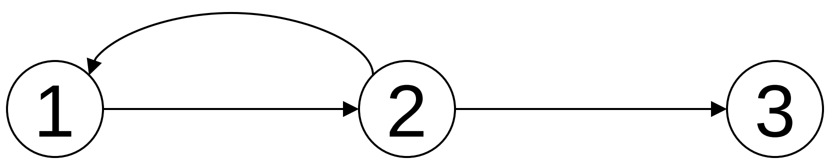
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

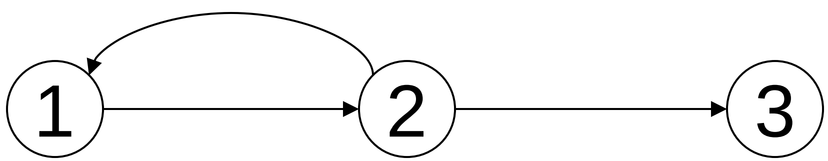
Example 2

$$r_2(A);\ r_1(B);\ w_2(A);\ r_2(B);\ r_3(A);\ w_1(B);\ w_3(A);\ w_2(B)$$



Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

Takeaways

- Transactions: “…all or nothing…”
- Simplified data model: READ/WRITE elements
- Schedules:
  - Serial
  - Serializable
  - Conflict serializable
- Precedence graph