



# Database Integrity in Django: Safely Handling Critical Data in Distributed Systems

Dealing with concurrency, money,  
and log-structured data in the Django ORM

Nick Sweeting (@theSquashSH)

Slides: [github.com/pirate/django-concurrency-talk](https://github.com/pirate/django-concurrency-talk)



# Background

Nick Sweeting

Twitter: theSquashSH | Github: pirate  
Co-Founder/CTO @ Monadical.com



We built OddSlingers.com, a fast, clean online poker experience made with Django + Channels & React/Redux. We learned a lot about database integrity along the way.

Disclaimer: I am not  
a distributed systems expert.

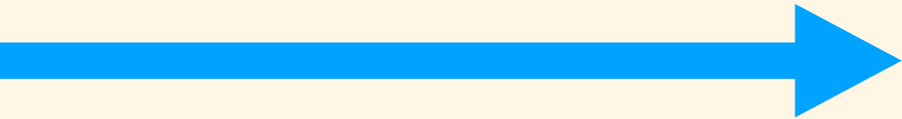
I just think they're neat.



It all starts with a single salami slice.



It ends with millions of dollars missing.



## **Dealing with money**

float, Decimal, and math

## **Avoiding concurrency**

linearizing writes in a queue

## **Dealing with concurrency**

transactions, locking, compare-and-swaps

## **Schema design**

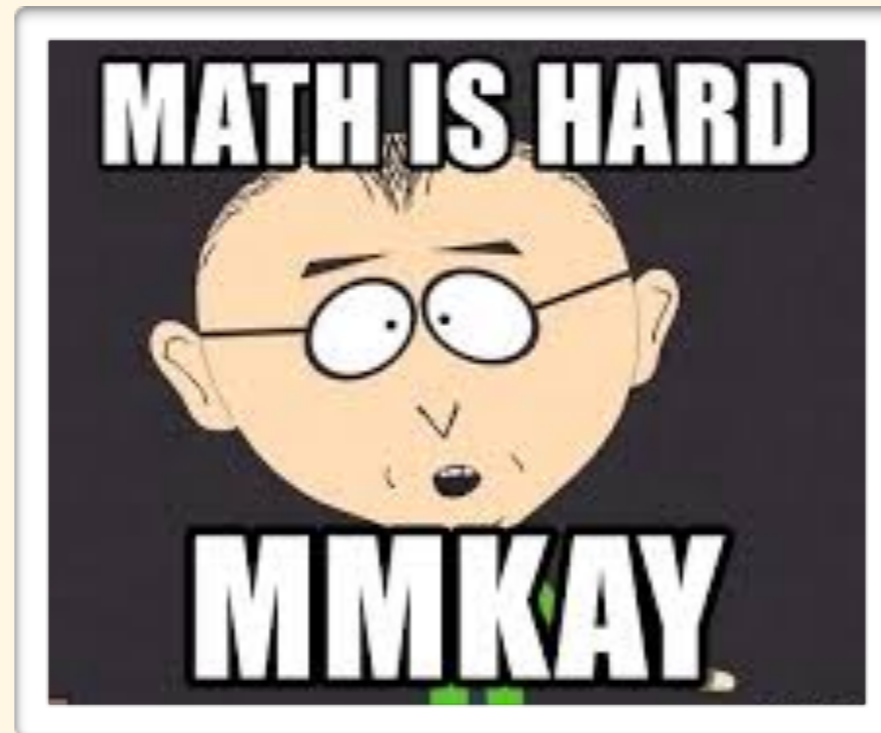
log-structured data, minimizing locks

## **The bigger picture**

code layout, storage layer, NewSQL databases

# *Dealing With Money*

float, Decimal, & math



# Losing track of fractional cents (aka salami slicing)

```
class Payout(models.Model):  
    ...  
    amt = models.DecimalField(max_digits=20, decimal_places=2)  
  
Payout.objects.create(amt=100000/7)  
# saves: 14285.71 instead of 14285.714285714
```

Do this 10 million times and suddenly you've lost track of  $0.0043 * 10m = \$42,857$ .



I know you're tempted,  
don't even try it...  
salami slicers all  
get caught eventually

# float vs Decimal

```
>>> 0.1 + 0.2  
0.30000000000000004
```

```
>>> 0.1 + 0.2 == 0.3  
False
```

Floating-point math introduces error that can accumulate.

```
>>> from decimal import Decimal  
>>> Decimal('0.1') + Decimal('0.2')  
Decimal('0.3')
```

Much better.

# Rounding in Python 3

```
>>> round(1.5)  
2
```

```
>>> round(2.5)  
2
```

**wat.**

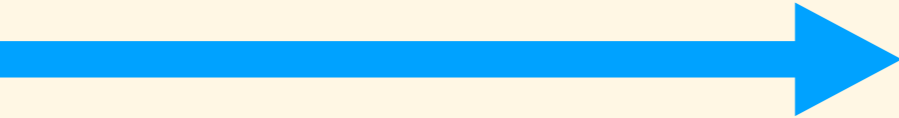
Get expected rounding behavior with:

```
>>> decimal.Decimal('2.5').quantize(decimal.Decimal('1'),  
                                     rounding=decimal.ROUND_HALF_UP)  
Decimal('3')
```



## **Dealing with money**

float, Decimal, and math



## **Avoiding concurrency**

linearizing writes in a queue

## **Dealing with concurrency**

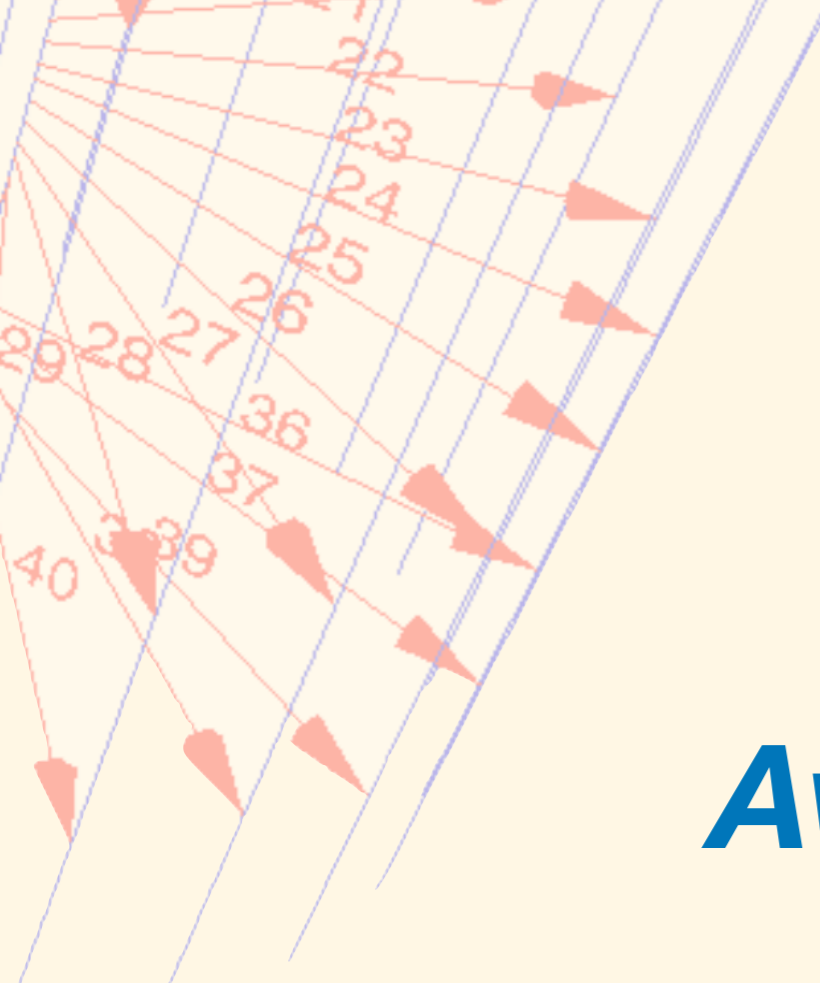
transactions, locking, compare-and-swaps

## **Schema design**

log-structured data, minimizing locks

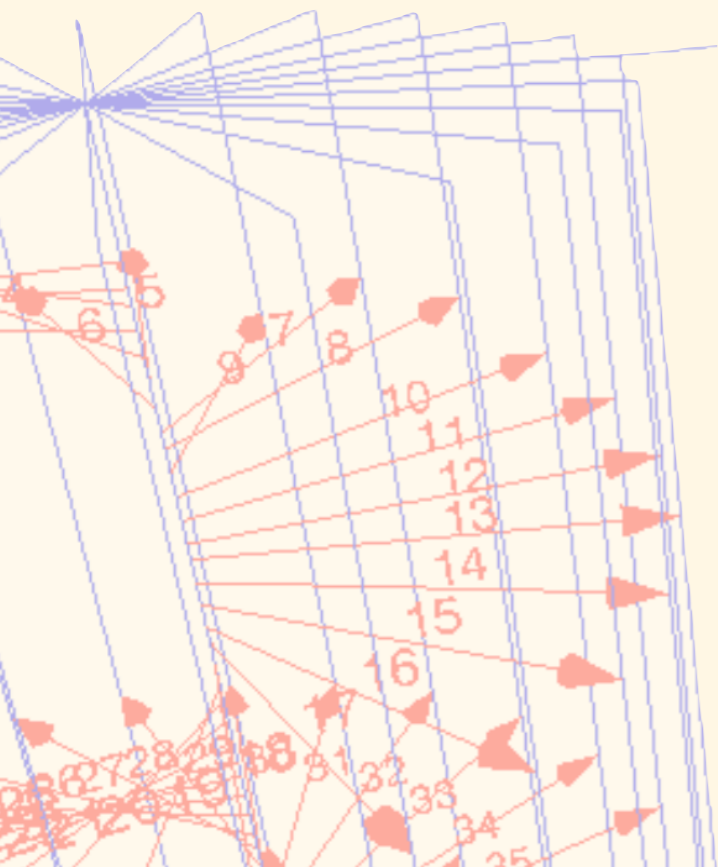
## **The bigger picture**

code layout, storage layer, NewSQL databases



# *Avoid Concurrency*

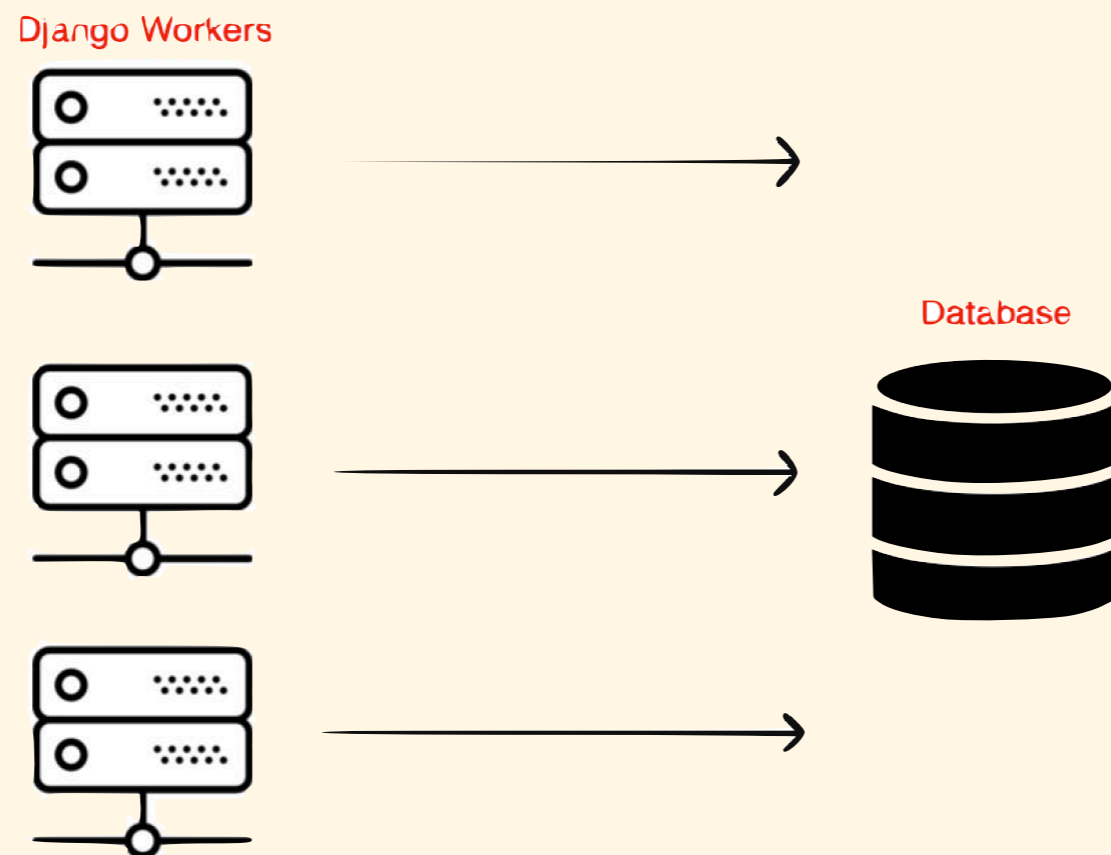
Eliminate the dragons.



# What is a distributed system?

Every Django app.

- > Each Django request handler is a separate thread
- > What happens when two threads try to do something critical at the same time? e.g. update a user's balance

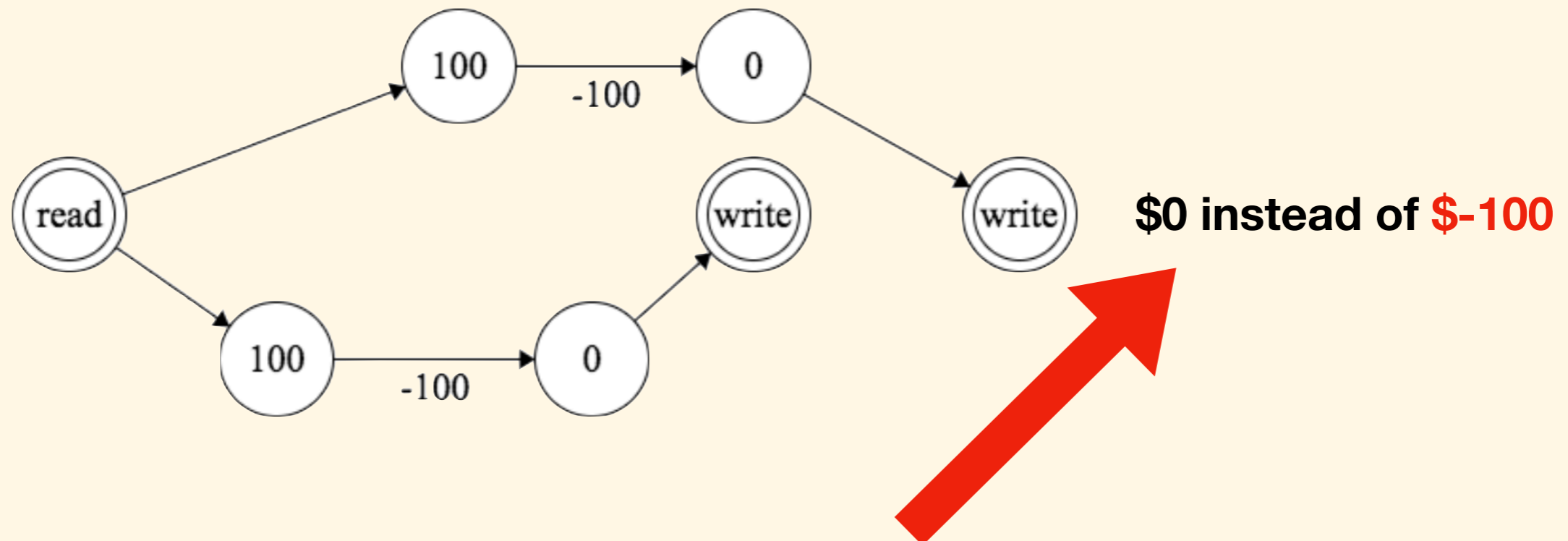


# Other threads writing will break the bank

Failure if two threads execute withdrawals at once:

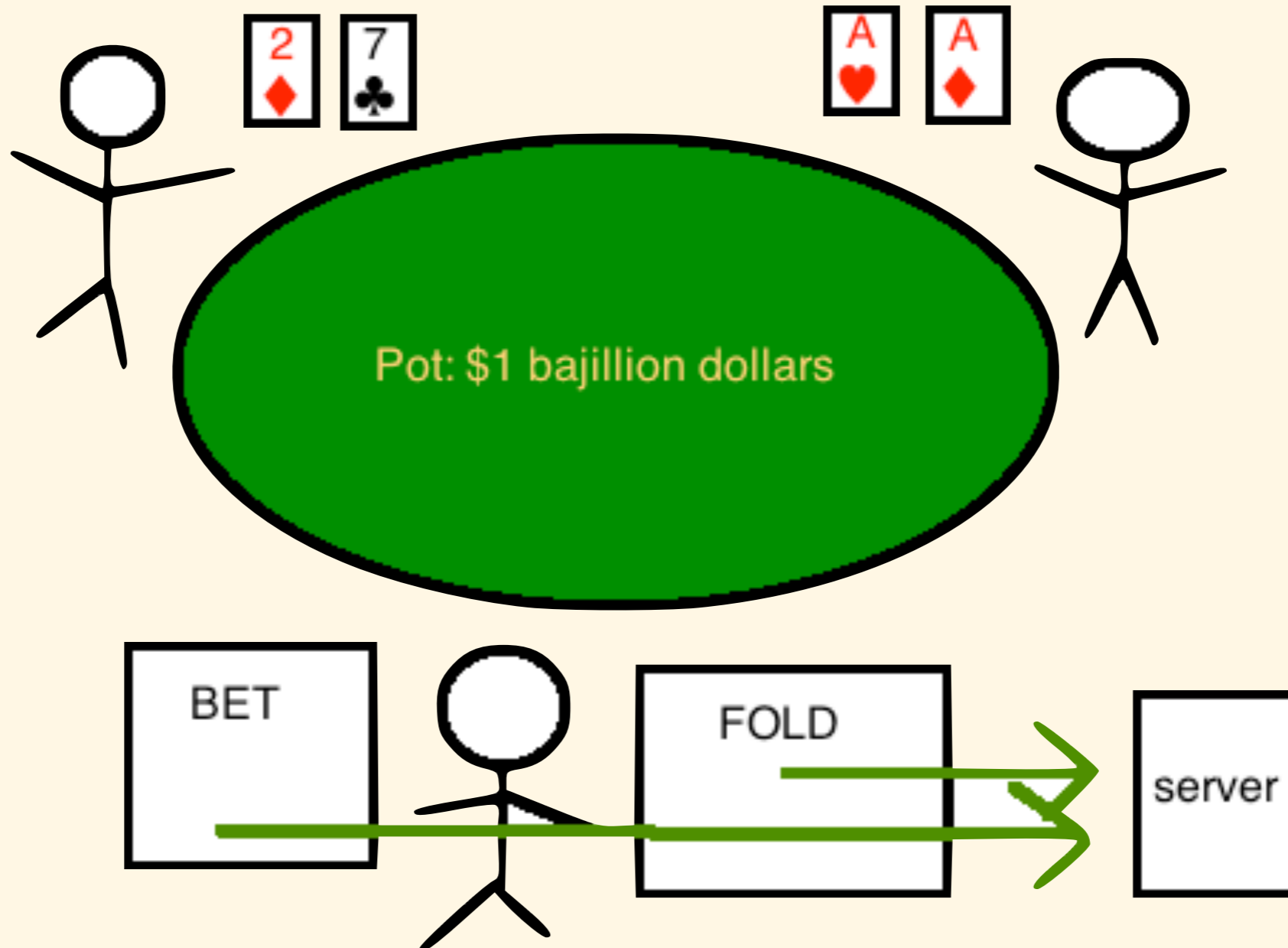
```
if (user.balance == 100) {  
    user.balance -= 100 # 2nd thread can .save() a 100 withdrawal befo  
    user.save()         # balance is now 0 but should be -100  
}
```

aka "Time of check to time of use"-bug (TOCTTOU).



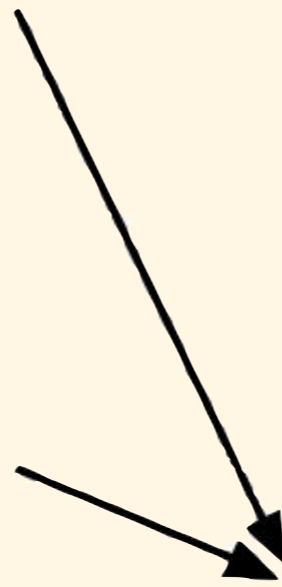
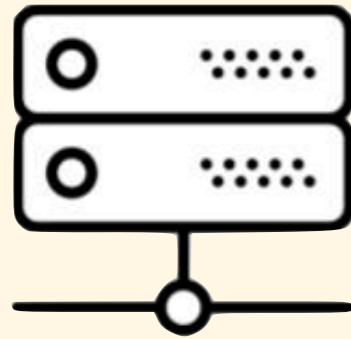
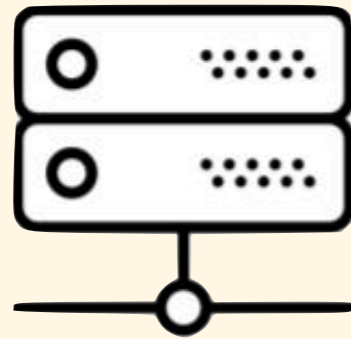
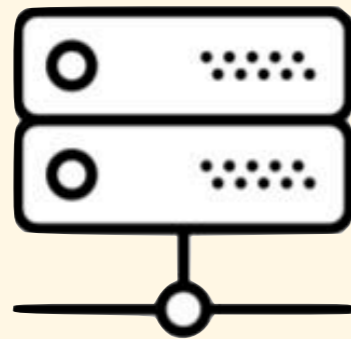
# The Challenge

Dealing with concurrent write conflicts

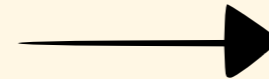


# A solution: remove the concurrency

Django Workers



Queue



Strictly order all state mutations by timestamp

Database



Linearize all the writes into a single queue

```
transactions = [  
    # timestamp      condition          action  
    (1523518620,     "can_deposit(241)",  "deposit_usd(241, 50)",  
    (1523518634,     "balance_gt(241, 50)", "buy_chips(241, 50)",  
]
```

**If only one change happens at a time,**

no conflicting writes can occur.

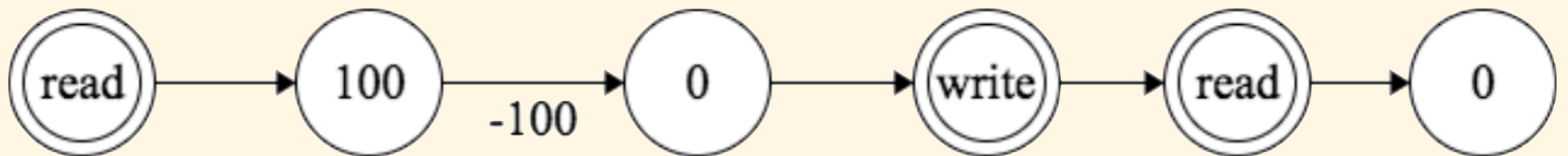
Execute the writes 1 by 1 in a dedicated process

(using a Redis Queue, or Dramatiq, Celery, etc.)

```
while True:
    ts, condition, action = transaction_queue.pop()
    if eval(condition):
        eval(action)
```

Don't let any other processes touch the same tables.

All checks & writes are now *linearized*.





# **Eliminate concurrency at all costs.**

If you value your sanity,  
linearize critical transactions into a single queue  
whenever possible.

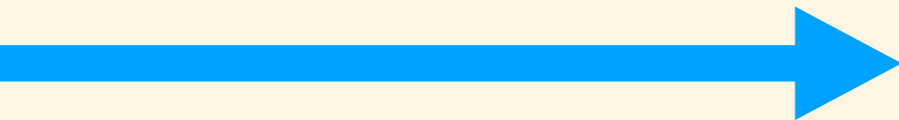
Don't even watch the rest of the talk,  
just stop now, really, you probably don't need concurrency...

## **Dealing with money**

float, Decimal, and math

## **Avoiding concurrency**

linearizing writes in a queue



## **Dealing with concurrency**

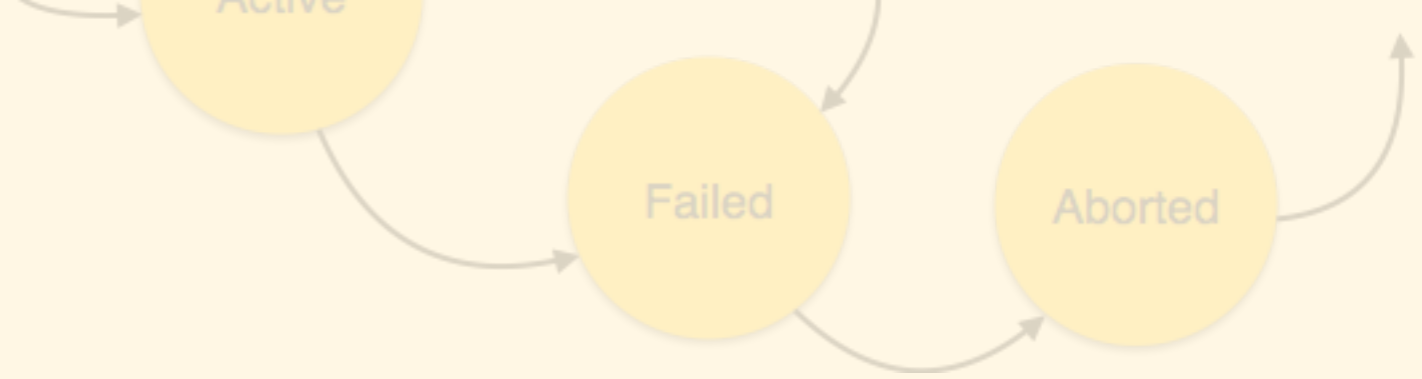
transactions, locking, compare-and-swaps

## **Schema design**

log-structured data, minimizing locks

## **The bigger picture**

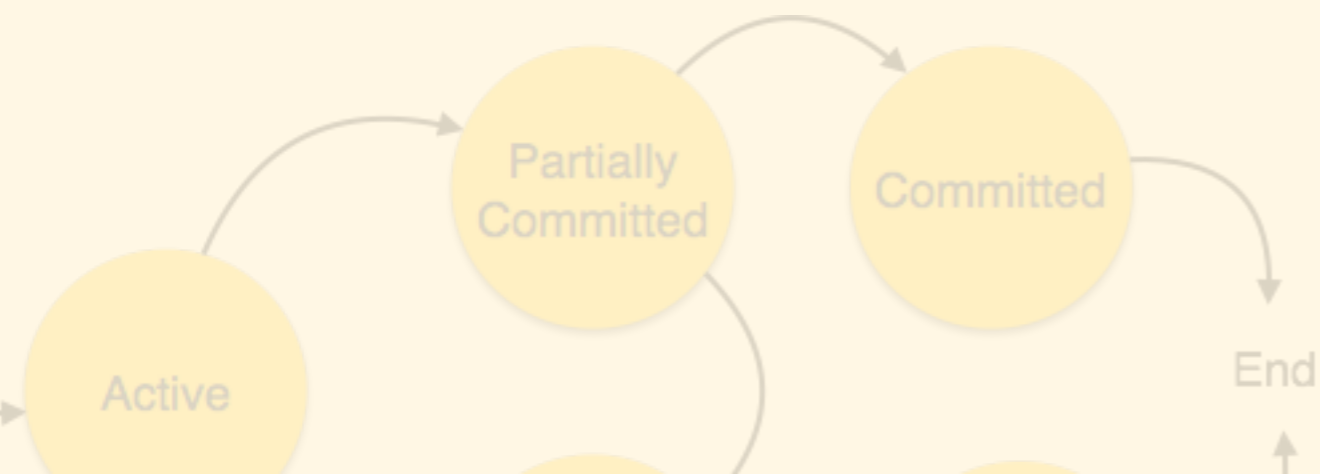
code layout, storage layer, NewSQL databases



# *Dealing With Concurrency*

transactions, locking, compare-and-swaps

I warned you about the dragons...



# Tools the ORM provides

> Atomic transactions

**transaction.atomic()**

> Locking

**Model.objects.select\_for\_update()**

> Compare-and-swaps

**.filter(val=expected).update(val=new)**

```
...
    elif isinstance(obj, basestring):
        if obj.startswith('http://') or obj.startswith('https://'):
            raise ValueError('URL requires a single quote to be taken care of.')
        return obj
    elif isinstance(obj, datetime.datetime):
        return obj.isoformat()
    elif isinstance(obj, decimal.Decimal):
        return obj.as_tuple()[0]
    elif isinstance(obj, float):
        return obj
    elif isinstance(obj, int):
        return obj
    elif isinstance(obj, list):
        return [self.serialize(item) for item in obj]
    elif isinstance(obj, tuple):
        return tuple(self.serialize(item) for item in obj)
    elif isinstance(obj, dict):
        return {key: self.serialize(value) for key, value in obj.items()}
    elif isinstance(obj, set):
        return list(self.serialize(item) for item in obj)
    elif isinstance(obj, frozenset):
        return list(self.serialize(item) for item in obj)
    elif isinstance(obj, bytes):
        return obj
    elif isinstance(obj, bytearray):
        return obj
    elif isinstance(obj, memoryview):
        return obj
    elif isinstance(obj, complex):
        return obj
    elif isinstance(obj, bool):
        return obj
    elif isinstance(obj, None):
        return None
    else:
        raise ValueError('Unsupported type: %s' % type(obj))

def serialize(obj):
    """Serialize an object to a string representation.

    This function is used to serialize objects for storage in a database.
    It handles various types of objects, including strings, integers,
    floats, lists, tuples, dictionaries, sets, frozensets, bytes,
    bytearray, memoryview, complex numbers, booleans, and None.

    The function returns a string representation of the object, which
    can be stored in a database. The string representation is a
    JSON-like format, but it is not a valid JSON string. It uses
    single quotes for strings and double quotes for other types.

    Example:
    serialize('http://example.com') returns 'http://example.com'
    serialize(123) returns '123'
    serialize(1.23) returns '1.23'
    serialize([1, 2, 3]) returns '[1, 2, 3]'
    serialize((1, 2, 3)) returns '(1, 2, 3)'
    serialize({'a': 1, 'b': 2}) returns '{"a": 1, "b": 2}'
    serialize({1, 2, 3}) returns '{1, 2, 3}'
    serialize(frozenset([1, 2, 3])) returns 'frozenset([1, 2, 3])'
    serialize(b'example') returns 'b"example"'
    serialize(bytearray(b'example')) returns 'bytearray(b"example")'
    serialize(memoryview(b'example')) returns 'memoryview(b"example")'
    serialize(1+2j) returns '(1+2j)'
    serialize(True) returns 'True'
    serialize(False) returns 'False'
    serialize(None) returns 'None'
    """
    if isinstance(obj, basestring):
        if obj.startswith('http://') or obj.startswith('https://'):
            raise ValueError('URL requires a single quote to be taken care of.')
        return obj
    elif isinstance(obj, datetime.datetime):
        return obj.isoformat()
    elif isinstance(obj, decimal.Decimal):
        return obj.as_tuple()[0]
    elif isinstance(obj, float):
        return obj
    elif isinstance(obj, int):
        return obj
    elif isinstance(obj, list):
        return [self.serialize(item) for item in obj]
    elif isinstance(obj, tuple):
        return tuple(self.serialize(item) for item in obj)
    elif isinstance(obj, dict):
        return {key: self.serialize(value) for key, value in obj.items()}
    elif isinstance(obj, set):
        return list(self.serialize(item) for item in obj)
    elif isinstance(obj, frozenset):
        return list(self.serialize(item) for item in obj)
    elif isinstance(obj, bytes):
        return obj
    elif isinstance(obj, bytearray):
        return obj
    elif isinstance(obj, memoryview):
        return obj
    elif isinstance(obj, complex):
        return obj
    elif isinstance(obj, bool):
        return obj
    elif isinstance(obj, None):
        return None
    else:
        raise ValueError('Unsupported type: %s' % type(obj))

def serialize(obj):
    """Serialize an object to a string representation.

    This function is used to serialize objects for storage in a database.
    It handles various types of objects, including strings, integers,
    floats, lists, tuples, dictionaries, sets, frozensets, bytes,
    bytearray, memoryview, complex numbers, booleans, and None.


    The function returns a string representation of the object, which
    can be stored in a database. The string representation is a
    JSON-like format, but it is not a valid JSON string. It uses
    single quotes for strings and double quotes for other types.

    Example:
    serialize('http://example.com') returns 'http://example.com'
    serialize(123) returns '123'
    serialize(1.23) returns '1.23'
    serialize([1, 2, 3]) returns '[1, 2, 3]'
    serialize((1, 2, 3)) returns '(1, 2, 3)'
    serialize({'a': 1, 'b': 2}) returns '{"a": 1, "b": 2}'
    serialize({1, 2, 3}) returns '{1, 2, 3}'
    serialize(frozenset([1, 2, 3])) returns 'frozenset([1, 2, 3])'
    serialize(b'example') returns 'b"example"'
    serialize(bytearray(b'example')) returns 'bytearray(b"example")'
    serialize(memoryview(b'example')) returns 'memoryview(b"example")'
    serialize(1+2j) returns '(1+2j)'
    serialize(True) returns 'True'
    serialize(False) returns 'False'
    serialize(None) returns 'None'
    """
    if isinstance(obj, basestring):
        if obj.startswith('http://') or obj.startswith('https://'):
            raise ValueError('URL requires a single quote to be taken care of.')
        return obj
    elif isinstance(obj, datetime.datetime):
        return obj.isoformat()
    elif isinstance(obj, decimal.Decimal):
        return obj.as_tuple()[0]
    elif isinstance(obj, float):
        return obj
    elif isinstance(obj, int):
        return obj
    elif isinstance(obj, list):
        return [self.serialize(item) for item in obj]
    elif isinstance(obj, tuple):
        return tuple(self.serialize(item) for item in obj)
    elif isinstance(obj, dict):
        return {key: self.serialize(value) for key, value in obj.items()}
    elif isinstance(obj, set):
        return list(self.serialize(item) for item in obj)
    elif isinstance(obj, frozenset):
        return list(self.serialize(item) for item in obj)
    elif isinstance(obj, bytes):
        return obj
    elif isinstance(obj, bytearray):
        return obj
    elif isinstance(obj, memoryview):
        return obj
    elif isinstance(obj, complex):
        return obj
    elif isinstance(obj, bool):
        return obj
    elif isinstance(obj, None):
        return None
    else:
        raise ValueError('Unsupported type: %s' % type(obj))

```

# Atomic Transactions

```
with transaction.atomic():  
    thing = SomeModel.objects.create(...)  
    other_thing = SomeModel.objects.create(...)  
  
    if error_condition(...):  
        raise Exception('Rolls back entire transaction')
```



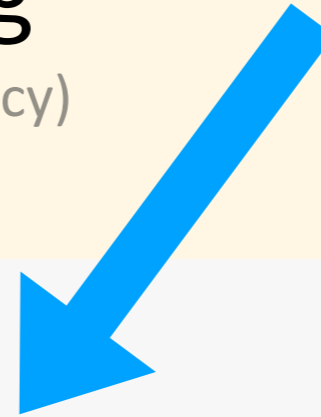
Exceptions roll back the entire transaction block.

Neither object will be saved to the DB.

Transactions can be nested.

# Row Locking

(pessimistic concurrency)



```
with transaction.atomic():  
    to_update = SomeModel.objects.select_for_update().filter(id=thing.id)  
    ...  
    to_update.update(val=new)
```

`.select_for_update()` allows you to lock rows

Locking prevents other threads from changing the row until the end of the current transaction, when the lock is released.

# Atomic compare-and-swaps

(optimistic concurrency)

```
last_changed = obj.modified
```

```
...
```

```
SomeModel.objects.filter(id=obj.id, modified=last_changed).update(val=new_val)
```

Only updates if the db row is unchanged by other threads.

- > any modified obj in db will differ from our stale in-memory obj ts
- > filter() wont match any rows, update() fails
- > overwriting newer row in db with stale data is prevented

This is very hard to get right, locking is better for 90% of use cases!

# Hybrid Solution

(optimistic concurrency + pessimistic or Multiversion Concurrency Control)

```
last_changed = obj.modified
```

```
... read phase
```

```
SomeModel.objects.select_for_update().filter(id=obj.id, modified=last_changed)
```

```
... write phase
```

## Best of both worlds

- > locking is limited to write-phase only
- > no need for complex multi-model compare-and-swaps

MVCC is used internally by PostgreSQL

Alternative: SQL gap-locking w/ filter query on indexed col.



## **Dealing with money**

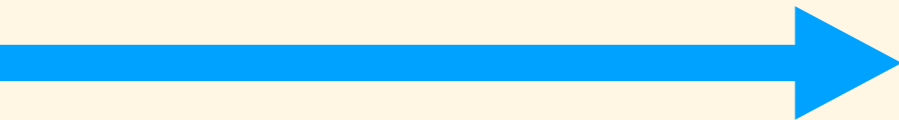
float, Decimal, and math

## **Avoiding concurrency**

linearizing writes in a queue

## **Dealing with concurrency**

transactions, locking, compare-and-swaps



## **Schema design**

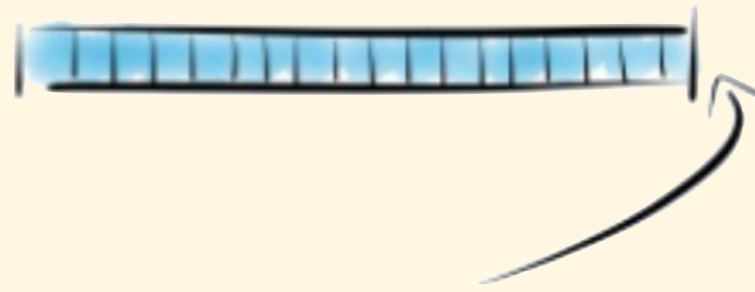
log-structured data, minimizing locks

## **The bigger picture**

code layout, storage layer, NewSQL databases

# Schema Design

log-structured data, minimizing locks



Append Only

# What is log-structured data?

Append-only tables vs mutable tables

## > Mutable example

User	Balance
Alice	52
Bob	21

## > Log-structured example (immutable, append-only)

Timestamp	User	Transfer
15000000001	Alice	+100
15000000002	Alice	- 50
15000000003	Bob	+ 2

Totals are derived from sum of rows for a given user

# **Log-structured storage is a foundational building block of safe, distributed systems.**

- Provides strict ordering of writes
- Immutable log of every change
- Ability to revert to any point in time

See: [redux](#), [CouchDB](#), [Redis](#)

But log-structured tables make  
locking hard...

Because any new row added can change the total,  
we'd have to lock the entire BalanceTransfer  
table to prevent concurrent processes from  
adding new transfers that change the total.

How else can we prevent  
concurrent writes from changing a  
user's balance?

Store a total separately from the log,  
require they be updated together

```
class UserBalance(models.Model):  
    user = models.OneToOneField(User)  
    total = models.DecimalField(max_digits=20, decimal_places=2)
```

A single-row lock must now be obtained on the **total** before adding new BalanceTransfer rows for that user.

# Full example using locking

```
def send_money(src, dst, amt):  
    with transaction.atomic():  
        # Lock balance rows, preventing other threads from making changes  
        src_bal = UserBalance.objects.select_for_update().filter(id=src)  
        dst_bal = UserBalance.objects.select_for_update().filter(id=dst)  
  
        if src_bal[0].total < amt:  
            raise Exception('Not enough balance to complete transaction')  
  
        # Update the totals and add a BalanceTransfer log row together  
        BalanceTransfer.objects.create(src=src, dst=dst, amt=amt)  
        src_bal.update(total=F('total') - amt)  
        dst_bal.update(total=F('total') + amt)
```

Side benefit: no need to scan entire BalanceTransfer table anymore to get a user's balance

# **Log-structured data is great, but...**

it requires careful thought to:

- minimize detrimental whole-table locking
- access aggregate values without scanning



## **Dealing with money**

float, Decimal, and math

## **Avoiding concurrency**

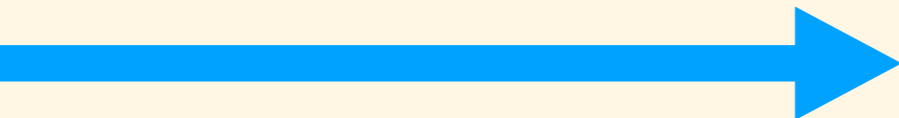
linearizing writes in a queue

## **Dealing with concurrency**

transactions, locking, compare-and-swaps

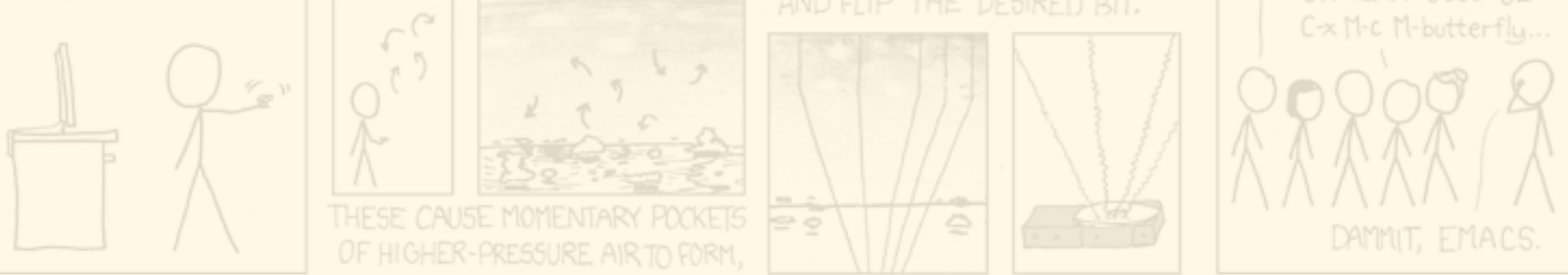
## **Schema design**

log-structured data, minimizing locks



## **The bigger picture**

code layout, storage layer, NewSQL databases



## *The bigger picture*

code layout, storage layer, NewSQL databases

What happens when the butterflies flip your bits?



# Code Layout

- > Only perform writes via helper funcs, never update models directly
- > Put all transactions in one file for easier auditing & testing

```
banking/transactions.py:  
  
def transfer_money(src, dst, amt):  
    with transaction.atomic():  
        ...  
  
def merge_accounts(user_a, user_b):  
    with transaction.atomic():  
        ...  
  
def archive_account(user):  
    with transaction.atomic():  
        ...
```

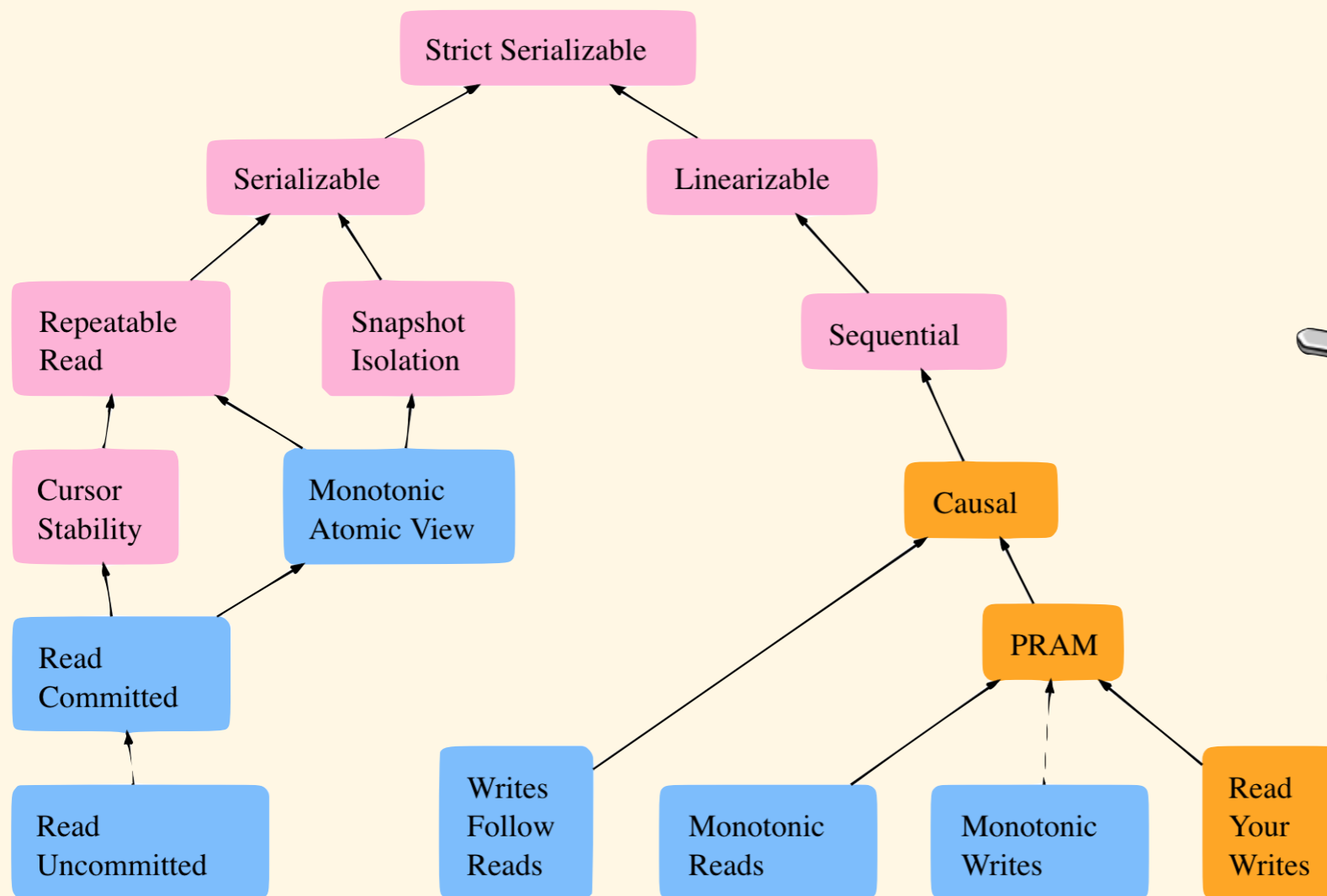
```
from banking.transactions import transfer_money  
  
...
```



# Database Isolation Levels

In some modes, partial transaction state can leak into other threads.

```
DATABASES = {  
    'OPTIONS': {  
        'isolation_level': psycopg2.extensions.ISOLATION_LEVEL_SERIALIZABLE,
```



Highly complex topic, much more info can be found elsewhere...

It's possible to use a separate database with a higher isolation level for critical data

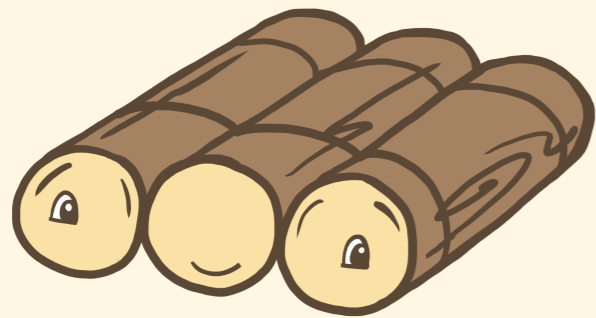
```
with transaction.atomic(using='default'):  
    with transaction.atomic(using='banking'):  
        MyModel_one(...).save(using='default')  
        MyModel_two(...).save(using='banking')
```

Django supports transactions across multiple databases.

# What the Future Looks Like

Serializable, distributed SQL without sharding.

SQL on top of >  
key:val store on top of >  
raft-based log-structured storage



CockroachDB & TiDB work with Python

## **Dealing with money**

float, Decimal, and math

## **Avoiding concurrency**

linearizing writes in a queue

## **Dealing with concurrency**

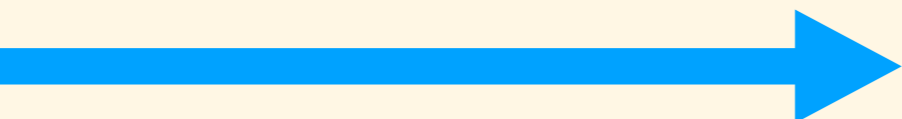
transactions, locking, compare-and-swaps

## **Schema design**

log-structured data, minimizing locks

## **The bigger picture**

code layout, storage layer, NewSQL databases



*The End*



# Key takeaways: don't stop worrying, but love atomic()

Don't use floats.

Don't use round(), if you must, always account for the remainder.

Don't execute non thread-safe writes in a parallel environment.

Use Decimal instead of float, and Decimal.quantize() instead of round() :

```
Decimal('0.35') + Decimal('100.15')
```

Lock dependent rows during transactions:

```
with transaction.atomic():  
    players = Player.objects.filter(user=user).select_for_update()  
    user.balance = player_balance_sum(players)  
    user.save()
```

Use atomic compare-and-swap operations when you cant lock:

```
User.objects.filter(id=user.id, balance__gt=50)\  
    .update(balance=F('balance') - 50)`
```

# Q&A

*Special thanks to:*

Django Core Team & Contributors,  
PyGotham Organizers, Andrew Godwin,  
Aphyr, Tyler Neely

Final Disclaimer: I'm not qualified to tell you how to design your distributed system. Get a professional for that.

I can only show you challenges and solutions I've discovered in my personal adventures with Django. Please let me know if you have corrections!

**Monadical is hiring and taking investment right now!**

contact me: [talks@sweeting.me](mailto:talks@sweeting.me)

Slides & Info: [github.com/pirate/django-concurrency-talk](https://github.com/pirate/django-concurrency-talk)