Pessimism, optimism, realism

and Django database concurrency

Aivars Kalvāns | aivars.kalvans@gmail.com | https://aivarsk.com

Me

2002 - 2020 @ **TietoEvry** (Payment cards)

C++, **SQL**, Python, …

Hundreds and thousands of transactions per second

2021 - now @ **Ebury** (Forex, Payments)

Python, **ORM**, Kafka, Kubernetes, AWS, …

Hundreds of thousands of € per transaction

Account transfer

```
def transfer(from_account: Account, to_account: Account, amount: int):
    if from_account.balance < amount:
        raise InsufficientBalance()

    from_account.balance -= amount
    to_account.balance += amount
```

Multiple threads

```
def transfer(from_account: Account, to_account: Account, amount: int):
    if from_account.balance < amount:
        raise InsufficientBalance()

    from_account.balance -= amount
    to_account.balance += amount


[...]

threading.Thread(target=process_transfers).start()
```

Time-of-check to time-of-use

```
def transfer(from_account: Account, to_account: Account, amount: int):
    if from_account.balance < amount: ⇦⇦⇦
        raise InsufficientBalance()

    from_account.balance -= amount ⇦⇦⇦
    to_account.balance += amount


[...]

threading.Thread(target=process_transfers).start()
```

Lost updates

```
def transfer(from_account: Account, to_account: Account, amount: int):
    if from_account.balance < amount:
        raise InsufficientBalance()

    from_account.balance -= amount ⇦⇦⇦
    to_account.balance += amount ⇦⇦⇦


[...]

threading.Thread(target=process_transfers).start()
```

Lost updates in detail

```
13          18 LOAD_FAST            0 (from_account)
            20 DUP_TOP
            22 LOAD_ATTR            0 (balance) ⇦⇦⇦
            24 LOAD_FAST            2 (amount)
            26 INPLACE_SUBTRACT
            28 ROT_TWO
            30 STORE_ATTR           0 (balance) ⇦⇦⇦

14          32 LOAD_FAST            1 (to_account)
            34 DUP_TOP
            36 LOAD_ATTR            0 (balance) ⇦⇦⇦
            38 LOAD_FAST            2 (amount)
            40 INPLACE_ADD
            42 ROT_TWO
            44 STORE_ATTR           0 (balance) ⇦⇦⇦
```

Shared memory concurrency

Concurrent components communicate by altering the contents of shared memory locations

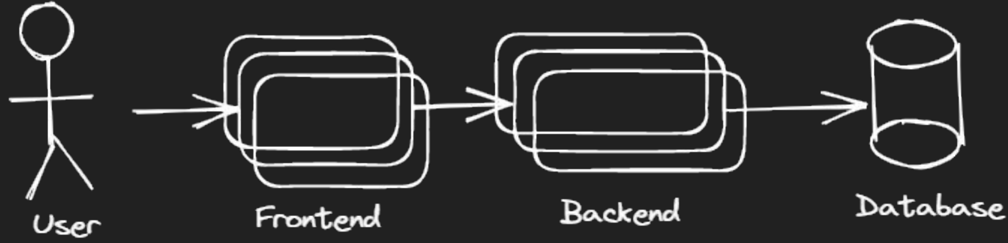
- Mutex, Futex, Read-write lock, Spinlock, Latch, Semaphore
- Thread-safe, Lock-free datastructures
- Atomic operations, Compare-and-Swap (CAS), Read-Copy-Update (RCU)
- Coroutines, Actor model

Python locks

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with from_account.lock, to_account.lock:
        if from_account.balance < amount:
            raise InsufficientBalance()

        from_account.balance -= amount
        to_account.balance += amount
```

A reusable architecture diagram



Back to where we started

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with transaction.atomic():
        if from_account.balance < amount:
            raise InsufficientBalance()

        from_account.balance -= amount
        to_account.balance += amount

        from_account.save()
        to_account.save()


transfer(Account.objects.get(pk=from_iban), Account.objects.get(pk=to_iban), 10)
```

Examples…

Flexcoin was a bitcoin exchange based in Alberta, Canada. The company was forced to shut down in March 2014 after hackers stole 896 units of the digital currency from its bitcoin bank.

The attacker then successfully exploited a flaw in the code which allows transfers between flexcoin users. By sending thousands of simultaneous requests, the **attacker was able to "move" coins from one user account to another** until the sending account was overdrawn, **before balances were updated**.

https://web.archive.org/web/20140305135801/http://flexcoin.com/

Feral Concurrency Control:
An Empirical Investigation of Modern Application Integrity

[...] The failures we observe here are solely due to concurrent execution. Without concurrent execution, validations are correct. [...]

Shared database concurrency

Concurrent components communicate by altering the contents of shared database records

- Pessimistic locking
- Optimistic locking
- ~~“Use Redis for locking”~~

Pessimistic locking

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with transaction.atomic():
        from_account = Account.objects.select_for_update().get(pk=from_account.pk)
        to_account = Account.objects.select_for_update().get(pk=to_account.pk)

        if from_account.balance < amount:
            raise InsufficientBalance()

        from_account.balance -= amount
        to_account.balance += amount

        from_account.save()
        to_account.save()
```

Optimistic locking

- A “version” field
  - number
  - timestamp
  - checksum - ABA problem
- Ensure the row has not changed since
- **Retries**

- django-concurrency
- django-optimistic-lock
- django-locking
- …

“avoids database-level locking”

Optimistic locking using .select_for_update()

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with transaction.atomic():

        if from_account.balance < amount:
            raise InsufficientBalance()

        from_locked = Account.objects.select_for_update().get(
            pk=from_account.pk, version=from_account.version
        )
        to_locked = Account.objects.select_for_update().get(
            pk=to_account.pk, version=to_account.version
        )

        from_account.balance -= amount
        from_account.version += 1
        to_account.balance += amount
        to_account.version += 1

        from_account.save()
        to_account.save()
```

Optimistic locking using .update()

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with transaction.atomic():
        if from_account.balance < amount:
            raise InsufficientBalance()

        count = Account.objects.filter(
            pk=from_account.pk, version=from_account.version
        ).update(
            balance=from_account.balance - amount, version=from_account.version + 1
        )
        if count != 1:
            raise TryAgain()

        count = Account.objects.filter(
            pk=to_account.pk, version=to_account.version
        ).update(
            balance=to_account.balance + amount, version=to_account.version + 1
        )
        if count != 1:
            raise TryAgain()
```

Avoiding database-level locking?

```
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.filter(
...     pk=from_account.pk, version=from_account.version
... ).update(
...     balance=from_account.balance - amount, version=from_account.version + 1
... )
1
>>>
```

```
>>>
```

Avoiding database-level locking?

```
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.filter(
...     pk=from_account.pk, version=from_account.version
... ).update(
...     balance=from_account.balance - amount, version=from_account.version + 1
... )
1
>>>
```

```
>>> Account.objects.filter(
...     pk=from_account.pk, version=from_account.version
... ).update(
...     balance=from_account.balance - amount, version=from_account.version + 1
... ) ⇦⇦⇦
```

Avoiding database-level locking?

```
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.filter(
...     pk=from_account.pk, version=from_account.version
... ).update(
...     balance=from_account.balance - amount, version=from_account.version + 1
... )
1
>>> transaction.commit()
>>>
```

```
>>> Account.objects.filter(
...     pk=from_account.pk, version=from_account.version
... ).update(
...     balance=from_account.balance - amount, version=from_account.version + 1
... )
0
>>>
```

Pessimistic vs Optimistic?

```
>>>
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> from_account = Account.objects.select_for_update().get(pk=from_account.pk)
>>>
```

```
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.filter(
...     pk=from_account.pk, version=from_account.version
... ).update(
...     balance=from_account.balance - amount, version=from_account.version + 1
... ) ⇦⇦⇦
```

Optimistic vs Pessimistic?

```
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.filter(
...     pk=from_account.pk, version=from_account.version
... ).update(
...     balance=from_account.balance - amount, version=from_account.version + 1
... )
1
>>>
```

```
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> from_account = Account.objects.select_for_update().get(pk=from_account.pk) ⇦⇦⇦
```

Pessimistic

```
def transfer(from_account: Account, to_account
    with transaction.atomic():
⇨⇨      from_account = Account.objects.select_
⇨⇨      to_account = Account.objects.select_fo

        if from_account.balance < amount:
            raise InsufficientBalance()

        from_account.balance -= amount
        to_account.balance += amount

        from_account.save()
        to_account.save()
```

Optimistic

```
def transfer(from_account: Account, to_acco
    with transaction.atomic():
        if from_account.balance < amount:
            raise InsufficientBalance()

        count = Account.objects.filter(
            pk=from_account.pk, version=fro
⇨⇨      ).update(
            balance=from_account.balance -
        )
        if count != 1:
            raise TryAgain()

        count = Account.objects.filter(
            pk=to_account.pk, version=to_ac
⇨⇨      ).update(
            balance=to_account.balance + am
        )
        if count != 1:
            raise TryAgain()
```

There is locking. Always!

- Explicit locking
- Implicit locking

Explicit locking and blocking

.select_for_update()

will block

.select_for_update(nowait=True)

does not block

.select_for_update(skip_locked=True)

does not block

Implicit locking and blocking

.update() and .save(force_update=True)

will block

.delete()

will block

.create() and .save(force_insert=True)

may block when unique constraint is being violated

QuerySet?

Blocking on .create()

```
>>>
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.create(pk=42)
<Account: Account object (42)>
>>>
```

```
>>>
>>>
```

Blocking on .create()

```
>>>
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.create(pk=42)
<Account: Account object (42)>
>>>
```

```
>>>
>>> Account.objects.create(pk=42) ⇦⇦⇦
```

Blocking on .create() until a .commit()

```
>>>
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.create(pk=42)
<Account: Account object (42)>
>>> transaction.commit()
>>>
```

```
>>>
>>> Account.objects.create(pk=42)
[...]
django.db.utils.IntegrityError: duplicate key value violates unique constraint
"app_account_pkey"
DETAIL:  Key (id)=(42) already exists.
>>>
```

Blocking on .create() until a .rollback()

```
>>>
>>> from django.db import transaction
>>> transaction.set_autocommit(False)
>>> Account.objects.create(pk=42)
<Account: Account object (42)>
>>> transaction.rollback()
>>>
```

```
>>>
>>> Account.objects.create(pk=42)
<Account: Account object (42)>
>>>
```

Locking and blocking everywhere

Improving concurrency

Python

- Increase granularity
- Hold locks for shorter duration

Database

- Fixed granularity - a row
- ???

Who releases the locks?

Row Locks (TX): A row lock, also called a TX lock, is a lock on a single row of a table. A transaction acquires a row lock for each row modified by using DML(INSERT, UPDATE, DELETE, MERGE) and SELECT ... FOR UPDATE. **The row lock exists until the transaction commits or rolls back.**

Improving concurrency

Python

- Increase granularity
- Hold locks for shorter duration

Database

- Fixed granularity - a row
- **Reduce time between update and commit**

Time between update and commit

```
with transaction.atomic():
    if from_account.balance < amount:
        raise InsufficientBalance()

    count = Account.objects.filter(
        pk=from_account.pk, version=from_account.version
⇨⇨ ).update(balance=from_account.balance - amount, version=from_account.version + 1)
    if count != 1:
        raise TryAgain()

    count = Account.objects.filter(
        pk=to_account.pk, version=to_account.version
    ).update(balance=to_account.balance + amount, version=to_account.version + 1)
    if count != 1:
        raise TryAgain()

    JournalEntry.objects.create(
        debit=from_account, credit=to_account, amount=amount
    )
⇨⇨
```

Time between update and commit

```
with transaction.atomic():
    if from_account.balance < amount:
        raise InsufficientBalance()

    JournalEntry.objects.create(
        debit=from_account, credit=to_account, amount=amount
    )

    count = Account.objects.filter(
        pk=from_account.pk, version=from_account.version
⇨⇨ ).update(balance=from_account.balance - amount, version=from_account.version + 1)
    if count != 1:
        raise TryAgain()

    count = Account.objects.filter(
        pk=to_account.pk, version=to_account.version
    ).update(balance=to_account.balance + amount, version=to_account.version + 1)
    if count != 1:
        raise TryAgain()
⇨⇨
```

Holding locks for shorter duration

- Do calculations early
- Move logging after transaction block
- Reorder database operations
- Update oversubscribed rows last
  - personal account first, bank account last
  - my twitter account first, Taylor Swift last
- Faster CPU, Network, Storage, etc.
- …

Oversubscribed rows

```
with transaction.atomic():
    if from_account.balance < amount:
        raise InsufficientBalance()

    count = Account.objects.filter(
        pk=from_account.pk, version=from_account.version
➪➪ ).update(balance=from_account.balance - amount, version=from_account.version + 1)
    # Blocks 10 potential updates
    if count != 1:
        raise TryAgain()

    count = Account.objects.filter(
        pk=to_account.pk, version=to_account.version
➪➪ ).update(balance=to_account.balance + amount, version=to_account.version + 1)
    # Blocks 10_000_000 potential updates
    if count != 1:
        raise TryAgain()
➪➪
```

Incrementing the version number

```
.update(balance=from_account.balance - amount, version=from_account.version + 1)
```

Incrementing the version number

```
.update(balance=from_account.balance - amount, version=from_account.version + 1)
```

```
UPDATE [...] SET [...], version = ?
```

Incrementing the version number

```
.update(balance=from_account.balance - amount, version=from_account.version + 1)
```

```
UPDATE [...] SET [...], version = ?
```

```
UPDATE [...] SET [...], version = version + 1
```

Incrementing the version number

```
.update(balance=from_account.balance - amount, version=from_account.version + 1)

UPDATE [...] SET [...], version = ?

UPDATE [...] SET [...], version = version + 1

.update(balance=from_account.balance - amount, version=F("version") + 1)
```

Incrementing the version number

```
.update(balance=from_account.balance - amount, version=from_account.version + 1)
```

```
UPDATE [...] SET [...], version = ?
```

```
UPDATE [...] SET [...], version = version + 1
```

```
.update(balance=from_account.balance - amount, version=F("version") + 1)
```

1. Load field value
2. Increment value
3. Store field value

What about lost updates?

```
for _ in range(1_000_000):
    from_account.version = F("version") + 1
    from_account.save()
```

How SQL UPDATE works

1. Filter rows according to WHERE clause
2. Lock rows: row-level locks
3. Evaluate SET clause: version = version + 1
4. Apply all changes

Update as a Python code

```
for _ in range(1_000_000):
    with from_account.lock
        from_account.version += 1
```

Commutative property

- Addition of both positive and negative numbers
  - counters of clicks, likes, votes, transactions
  - account balances
- Multiplication

Correct without pessimistic or optimistic locking

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with transaction.atomic():

        count = Account.objects.filter(
            pk=from_account.pk, balance_gte=amount
        ).update(balance=F("balance") - amount)
        if count != 1:
            raise InsufficientBalance()

        Account.objects.filter(pk=to_account.pk).update(
            balance=F("balance") + amount
        )
```

Retrieve the model after UPDATE

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with transaction.atomic():
        if from_account.balance < amount:
            raise InsufficientBalance()

        updated = Account.objects.raw( # https://code.djangoproject.com/ticket/32406
            """UPDATE "app_account" SET "balance" = "balance" - %s
                WHERE ("app_account"."id" = %s AND "app_account"."balance" >= %s)
            RETURNING * """,
            [ amount, from_account.pk, amount ],
        )
        try:
            from_account = updated[0]
        except IndexError:
            raise TryAgain()
        # ...
```

Using bulk operations

- .bulk_create()
- .bulk_update()

- PEP 249 and .executemany(operation, seq_of_parameters)
    - Prepare a database operation (query or command) and then execute it against all parameter sequences or mappings found in the sequence seq_of_parameters.

Correct using “bulk update” without explicit locking

```
def transfer(from_account: Account, to_account: Account, amount: int):
    with transaction.atomic():

        count = Account.objects.filter(
              Q(pk=from_account.pk, balance__gte=amount)
            | Q(pk=to_account.pk)
        ).update(
            balance=Case(
                When(pk=from_account.pk, then=F("balance") - amount),
                When(pk=to_account.pk,   then=F("balance") + amount),
            )
        )
        if count != 2:
            raise InsufficientBalance()
```

Takeaways

- Database **always** locks rows, leads to blocking
- **Accept it:** acquire as late as possible, release as soon as possible
- **Use it:** do locking in the database instead of Redis
- Choose and combine:
  - Order is not important: Implicit locking
  - Order is important, not know in advance: Optimistic
  - When optimistic fails: Pessimistic