Django ORM Mistakes

Kevin Mahoney
http://kevinmahoney.co.uk

January 20, 2017

Bug 1

```
def create():
    with transaction.atomic():
        thing = Thing.objects.create(foo=1, bar=2)
        set_foo(thing.id)
        thing.bar = 3
        thing.save()

def set_foo(id):
    thing = Thing.objects.get(id=id)
    thing.foo = 4
    thing.save()
```

Solution 1

```
def create():
    with transaction.atomic():
        thing = Thing.objects.create(foo=1, bar=2)
        set_foo(thing)
        thing.bar = 3
        thing.save()

def set_foo(thing):
    thing.bar = 4
    thing.save()
```

Bug 2

```
class Thing(Model):
    foo = ...
    bar = ...


def thing_set_foo(id, value):
    thing = Thing.objects.get(id=id)
    thing.foo = value
    thing.save()


def thing_set_bar(id, value):
    thing = Thing.objects.get(id=id)
    thing.bar = value
    thing.save()
```

Bug 2

$foo := 1, bar := 1$

T₁ T₂

$\texttt{READ}\ \ foo \rightarrow 1, bar \rightarrow 1$

$\texttt{ASSIGN}\ \ foo := 2$

$\texttt{WRITE}\ \ foo \leftarrow 2, bar \leftarrow 1$

$\texttt{READ}\ \ foo \rightarrow 1, bar \rightarrow 1$

$\texttt{ASSIGN}\ \ bar := 2$

$\texttt{WRITE}\ \ foo \leftarrow 1, bar \leftarrow 2$

Last write wins!

$foo \rightarrow 1, bar \rightarrow 2$

Solution 2a

```
def thing_set_foo(id, value):
    thing = Thing.objects.get(id=id)
    thing.foo = value
    thing.save(update_fields=["foo"])

def thing_set_bar(id, value):
    thing = Thing.objects.get(id=id)
    thing.bar = value
    thing.save(update_fields=["bar"])
```

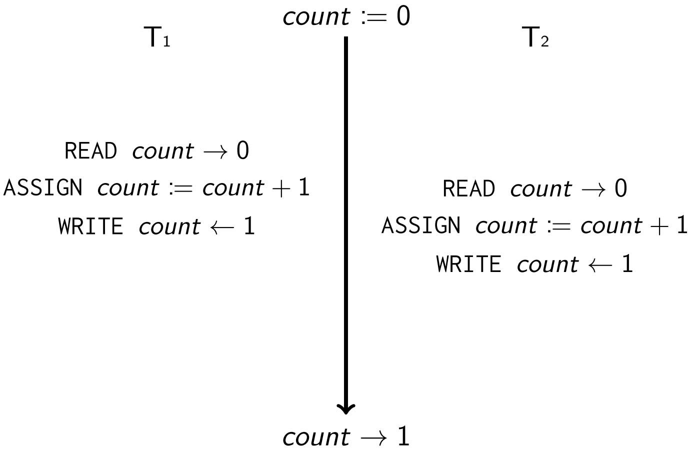
Solution 2b

```
def thing_set_foo(id, value):
    with transaction.atomic():
        thing = Thing.objects.select_for_update().get(id=id)
        thing.foo = value
        thing.save()

def thing_set_bar(id, value):
    with transaction.atomic():
        thing = Thing.objects.select_for_update().get(id=id)
        thing.bar = value
        thing.save()
```

Bug 3

```
def increment(id)
    counter = Counter.objects.get(id=id)
    counter.count = counter.count + 1
    counter.save()
```

Bug 3



$count := 0$

T_1

T_2

$\texttt{READ}\ \ count \rightarrow 0$

$\texttt{ASSIGN}\ \ count := count + 1$

$\texttt{WRITE}\ \ count \leftarrow 1$

$\texttt{READ}\ \ count \rightarrow 0$

$\texttt{ASSIGN}\ \ count := count + 1$

$\texttt{WRITE}\ \ count \leftarrow 1$

$count \rightarrow 1$

Solution 3

$count := 0$

T₁

~~READ $count \rightarrow 0$~~

~~ASSIGN $count := count + 1$~~

WRITE $count \leftarrow count + 1$

T₂

~~READ $count \rightarrow 0$~~

~~ASSIGN $count := count + 1$~~

WRITE $count \leftarrow count + 1$

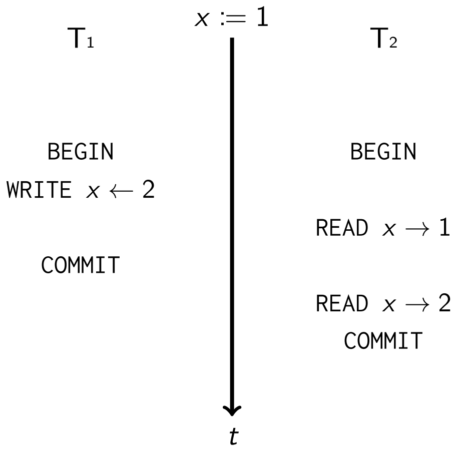
$count \rightarrow 2$

Solution 3

```
def increment(id)
    counter = Counter.objects.get(id=id)
    counter.count = F('count') + 1
    counter.save()
```

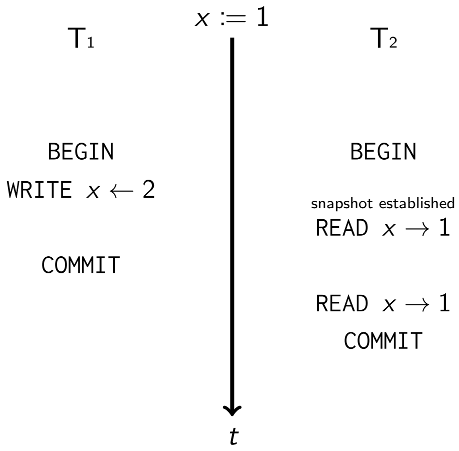
READ COMMITED Isolation

PostgreSQL's default isolation level.



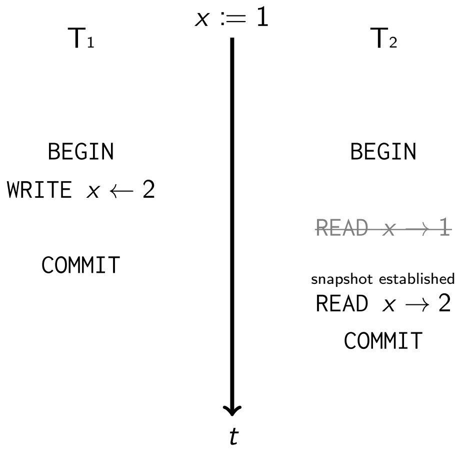
REPEATABLE READ Isolation

MySQL's default isolation level.



REPEATABLE READ Isolation

MySQL's default isolation level.



Bug 4

```
def create_payment(payment_collection, amount):
    with transaction.atomic():
        Payment.objects.create(amount=amount, payment_collection=payment_collection)
        payment_collection.total = (
            payment_collection.payment_set.all().aggregate(total=Sum('amount'))['total'])
        payment_collection.save()
```

Bug 4

$\textit{total} \rightarrow 0$

T_1	T_2
`BEGIN`	
`CREATE` $\textit{amount} := 6$	`BEGIN`
`READ` $\textit{total} \rightarrow 6$	`CREATE` $\textit{amount} := 10$
`WRITE` $\textit{total} \leftarrow 6$	`READ` $\textit{total} \rightarrow 10$
`COMMIT`	`WRITE` $\textit{total} \leftarrow 10$
	`COMMIT`

$\textit{total} \rightarrow 10$

Solution 4a

```
def create_payment(payment_collection, amount):
    with transaction.atomic():
        payment_collection = (PaymentCollection.objects
                                .select_for_update()
                                .get(id=payment_collection.id))
        Payment.objects.create(amount=amount, payment_collection=payment_collection)
        payment_collection.total = (
            payment_collection.payment_set.all().aggregate(total=Sum('amount'))['total'])
        payment_collection.save()
```

Solution 4b

```
def create_payment(payment_collection, amount):
    Payment.objects.create(amount=amount, payment_collection=payment_collection)
    with connection.cursor() as cursor:
        cursor.execute("""
        UPDATE payment_collection,
               (SELECT payment_collection_id, sum(amount) AS total
                FROM payment
                GROUP BY payment_collection__id) totals
        SET payment_collection.total = totals.total
        WHERE totals.pc_id = pc.id
          AND pc.id = %s
        """, [payment_collection.id])
```

Solution 4c

Even better (IMO)...

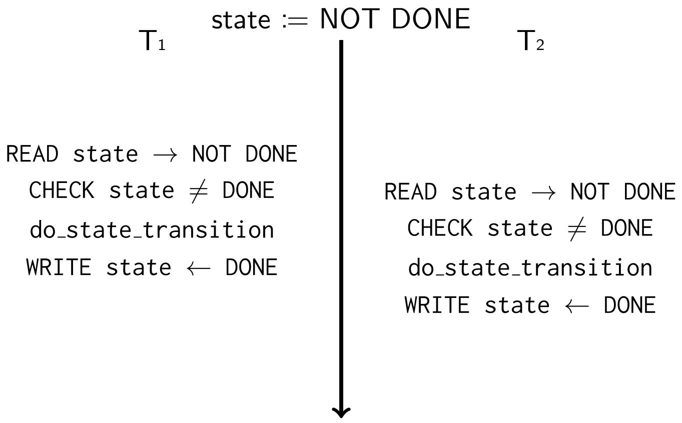
```
CREATE VIEW payment_collection_totals
  SELECT payment_collection_id, SUM(amount) AS total
  FROM payment
  GROUP BY payment_collection_id

CREATE VIEW payment_collection_with_total
  SELECT payment_collection.*, COALESCE(totals.total, 0) AS total
  FROM payment_collection
  LEFT JOIN totals ON (totals.payment_collection_id = payment_collection.id)
```

...but views as Python objects can awkward in Django.

Bug 5

```
def state_transition(id):
    with transaction.atomic():
        stateful = Stateful.objects.get(id=id)
        if stateful.state == DONE:
            raise AlreadyDone
        do_state_transition()
        stateful.state = DONE
        stateful.save()
```

Bug 5



```
READ state → NOT DONE
CHECK state ≠ DONE
do_state_transition
WRITE state ← DONE
```

```
READ state → NOT DONE
CHECK state ≠ DONE
do_state_transition
WRITE state ← DONE
```

Solution 5

```
def state_transition(id):
    with transaction.atomic():
        stateful = Stateful.objects.select_for_update().get(id=id)
        if stateful.state == DONE:
            raise AlreadyDone
        do_state_transition()
        stateful.state = DONE
        stateful.save()
```

Bug 6

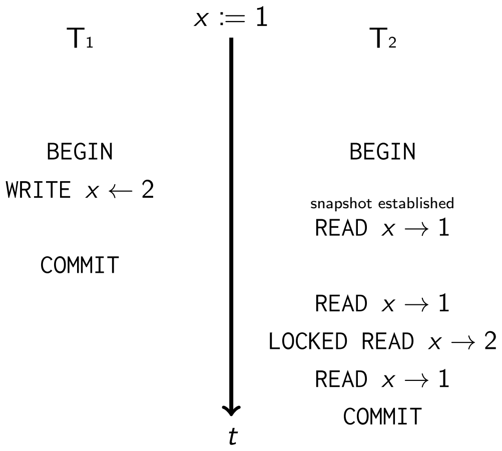
```
def foo(id):
    with transaction.atomic():
        thing = Thing.objects.get(id=id)
        result = OtherThing.objects.create(foo=thing.bar)

        with lock():
            thing.refresh_from_db()

            # If thing.other_thing has already been set in another
            # thread, raise an exception and rollback the transaction
            if thing.other_thing:
                raise Exception

            thing.other_thing = result
            thing.save()
```

Solution 6a

```
def foo(id):
    with lock():
        with transaction.atomic():
            thing = Thing.objects.get(id=id)
            result = OtherThing.objects.create(foo=thing.bar)

            # If thing.other_thing has already been set in another
            # thread, raise an exception and rollback the transaction
            if thing.other_thing:
                raise Exception

            thing.other_thing = result
            thing.save()
```

MySQL REPEATABLE READ Isolation

MySQL quirk!



Solution 6b

```
def foo(id):
    with transaction.atomic():
        thing = Thing.objects.get(id=id)
        result = OtherThing.objects.create(foo=thing.bar)

        # 'breaks' the snapshot established by the read above
        thing = Thing.objects.select_for_update().get(id=id)

        # If thing.other_thing has already been set in another
        # thread, raise an exception and rollback the transaction
        if thing.other_thing:
            raise Exception

        thing.other_thing = result
        thing.save()
```

Solution 6c

But this is better...

```
def foo(id):
    with transaction.atomic():
        thing = Thing.objects.select_for_update().get(id=id)
        result = OtherThing.objects.create(foo=thing.bar)

        # If thing.other_thing has already been set in another
        # thread, raise an exception and rollback the transaction
        if thing.other_thing:
            raise Exception

        thing.other_thing = result
        thing.save()
```

Bug 7

```
def increment(id)
    with transaction.atomic():
        counter = Counter.objects.get(id=id)
        counter.count = F('count') + 1
        counter.save()
```

Bug 7

```
def increment(id)
    with transaction.atomic():
        counter = Counter.objects.get(id=id)
        counter.count = F('count') + 1
        counter.save()
```

Not A Bug! (but confusing)

Reads in an update are locked! But, as before, it's probably confusing to rely on this behaviour and it won't work in PostgreSQL's REPEATABLE READ.

Back to Solution 4b

```
def create_payment(payment_collection, amount):
    Payment.objects.create(amount=amount, payment_collection=payment_collection)
    with connection.cursor() as cursor:
        cursor.execute("""
        UPDATE payment_collection,
               (SELECT payment_collection_id, sum(amount) AS total
                FROM payment
                GROUP BY payment_collection__id) totals
        SET payment_collection.total = totals.total
        WHERE totals.pc_id = pc.id
          AND pc.id = %s
        """, [payment_collection.id])
```

Solution 4d?

```
def create_payment(payment_collection, amount):
    with transaction.atomic():
        Payment.objects.create(amount=amount, payment_collection=payment_collection)
        with connection.cursor() as cursor:
            cursor.execute("""
            UPDATE payment_collection,
                   (SELECT payment_collection_id, sum(amount) AS total
                    FROM payment
                    GROUP BY payment_collection__id) totals
            SET payment_collection.total = totals.total
            WHERE totals.pc_id = pc.id
              AND pc.id = %s
            """, [payment_collection.id])
```

Solution 4d?

```
def create_payment(payment_collection, amount):
    with transaction.atomic():
        Payment.objects.create(amount=amount, payment_collection=payment_collection)
        with connection.cursor() as cursor:
            cursor.execute("""
            UPDATE payment_collection,
                   (SELECT payment_collection_id, sum(amount) AS total
                    FROM payment
                    GROUP BY payment_collection__id) totals
            SET payment_collection.total = totals.total
            WHERE totals.pc_id = pc.id
              AND pc.id = %s
            """, [payment_collection.id])
```

Deadlock!

Tips

Tips

- Remember the ORM is an in-memory cache

Tips

- Remember the ORM is an in-memory cache
- ORMs can obscure bugs. Look at the SQL!

Tips

- Remember the ORM is an in-memory cache
- ORMs can obscure bugs. Look at the SQL!
- Avoid read-modify-write
  - If you don't you'll probably need a lock

Tips

- Remember the ORM is an in-memory cache
- ORMs can obscure bugs. Look at the SQL!
- Avoid read-modify-write
  - If you don't you'll probably need a lock
- If it's not locked, it's not up to date.

Tips

- Remember the ORM is an in-memory cache
- ORMs can obscure bugs. Look at the SQL!
- Avoid read-modify-write
  - If you don't you'll probably need a lock
- If it's not locked, it's not up to date.
- Lock before reads to avoid weird MySQL behaviour

Tips

- Remember the ORM is an in-memory cache
- ORMs can obscure bugs. Look at the SQL!
- Avoid read-modify-write
  - If you don't you'll probably need a lock
- If it's not locked, it's not up to date.
- Lock before reads to avoid weird MySQL behaviour
- Locking reads don't use the snapshot in MySQL

Tips

- Remember the ORM is an in-memory cache
- ORMs can obscure bugs. Look at the SQL!
- Avoid read-modify-write
  - If you don't you'll probably need a lock
- If it's not locked, it's not up to date.
- Lock before reads to avoid weird MySQL behaviour
- Locking reads don't use the snapshot in MySQL
- Prefer immutable database design if practical
  - e.g. 'event sourcing' style

Tips

- Remember the ORM is an in-memory cache
- ORMs can obscure bugs. Look at the SQL!
- Avoid read-modify-write
  - If you don't you'll probably need a lock
- If it's not locked, it's not up to date.
- Lock before reads to avoid weird MySQL behaviour
- Locking reads don't use the snapshot in MySQL
- Prefer immutable database design if practical
  - e.g. 'event sourcing' style
- Consider using a serializable isolation level
  - You don't have to worry about any of this if you use serializable transactions
  - Has other drawbacks
  - PostgreSQL implementation is nicer than the MySQL one IMO