The Ins & Outs of Model Inheritance

Blythe J Dunham
blythe@snowgiraffe.com

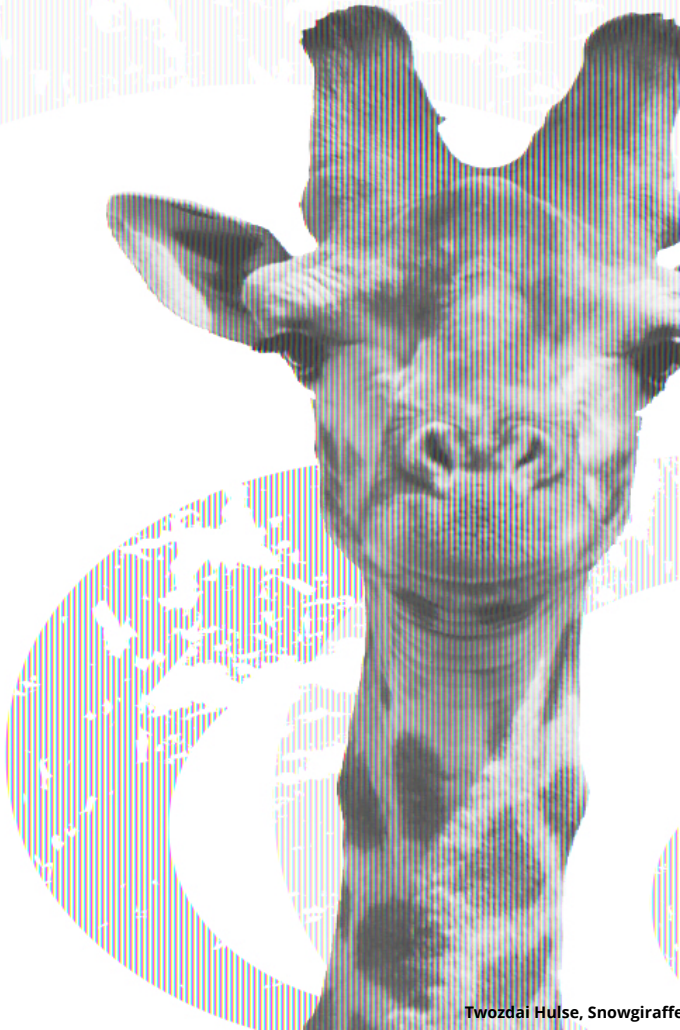
Today's Adventure Includes

- Composition vs Inheritance
- Three Types of Model Inheritance
  - Abstract Models (The Hum of the Giraffe)
  - Multi Table Inheritance (Don't Get Eaten by the Lion)
  - Proxy Models and Single Table Inheritance (Adventures in Middle Earth)
- Two Alternatives to Model Inheritance
- Avoiding Model Inheritance

Twozdai Hulse, Snowgiraffe

Today's Adventure Includes

- Composition vs Inheritance
- Three Types of Model Inheritance
  - Abstract Models (The Hum of the Giraffe)
  - Multi Table Inheritance (Don't Get Eaten by the Lion)
  - Proxy Models and Single Table Inheritance (Adventures in Middle Earth)
- Two Alternatives to Model Inheritance
- Avoiding Model Inheritance



Twozdai Hulse, Snowgiraffe

“Favor Composition Over Inheritance”

Design Patterns: Elements of Reusable Object-Oriented Software, 1994

Composition: “has-a”

- A giraffe has a (blue) tongue
- A hobbit has hairy feet
- A lion has big mouth full of teeth

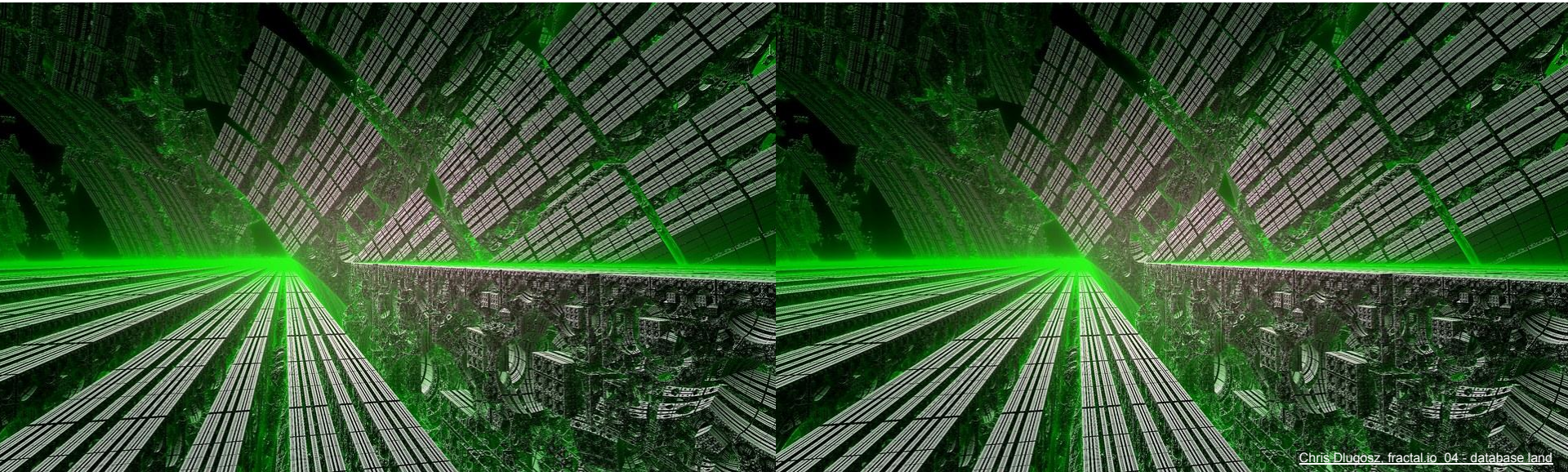
Eric Kilby, Closeup giraffe with tongue

Inheritance: “is-a”

- A giraffe is a (glorious) animal
- A lion is a big cat
- An elf is an immortal being

Relational Databases Don’t Implement Inheritance

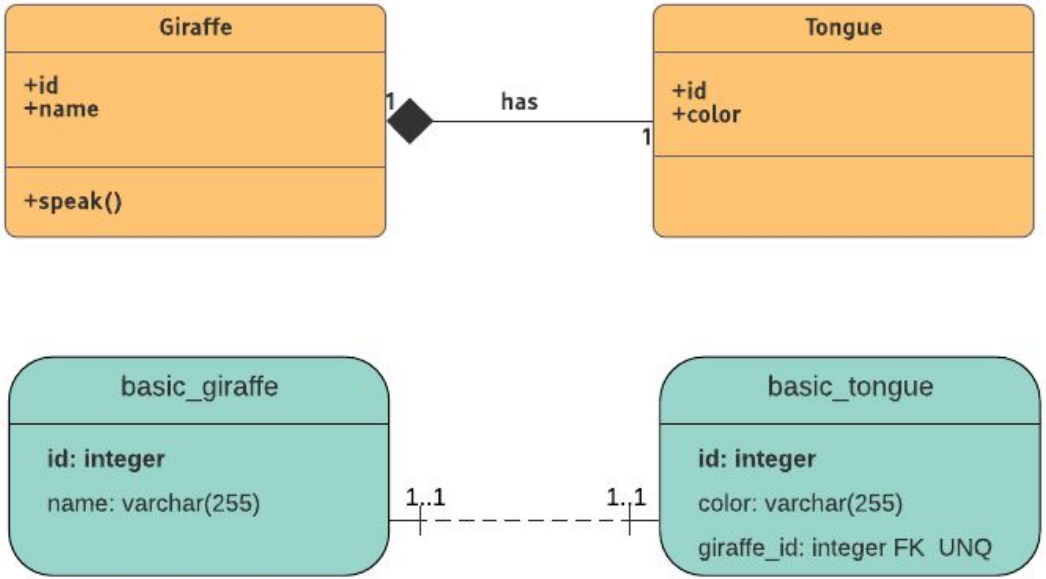
Because most RDBS aren’t object-oriented they do not support inheritance. While composition has a natural mapping, inheritance is not intuitive.



Django 101: Modeling Composition

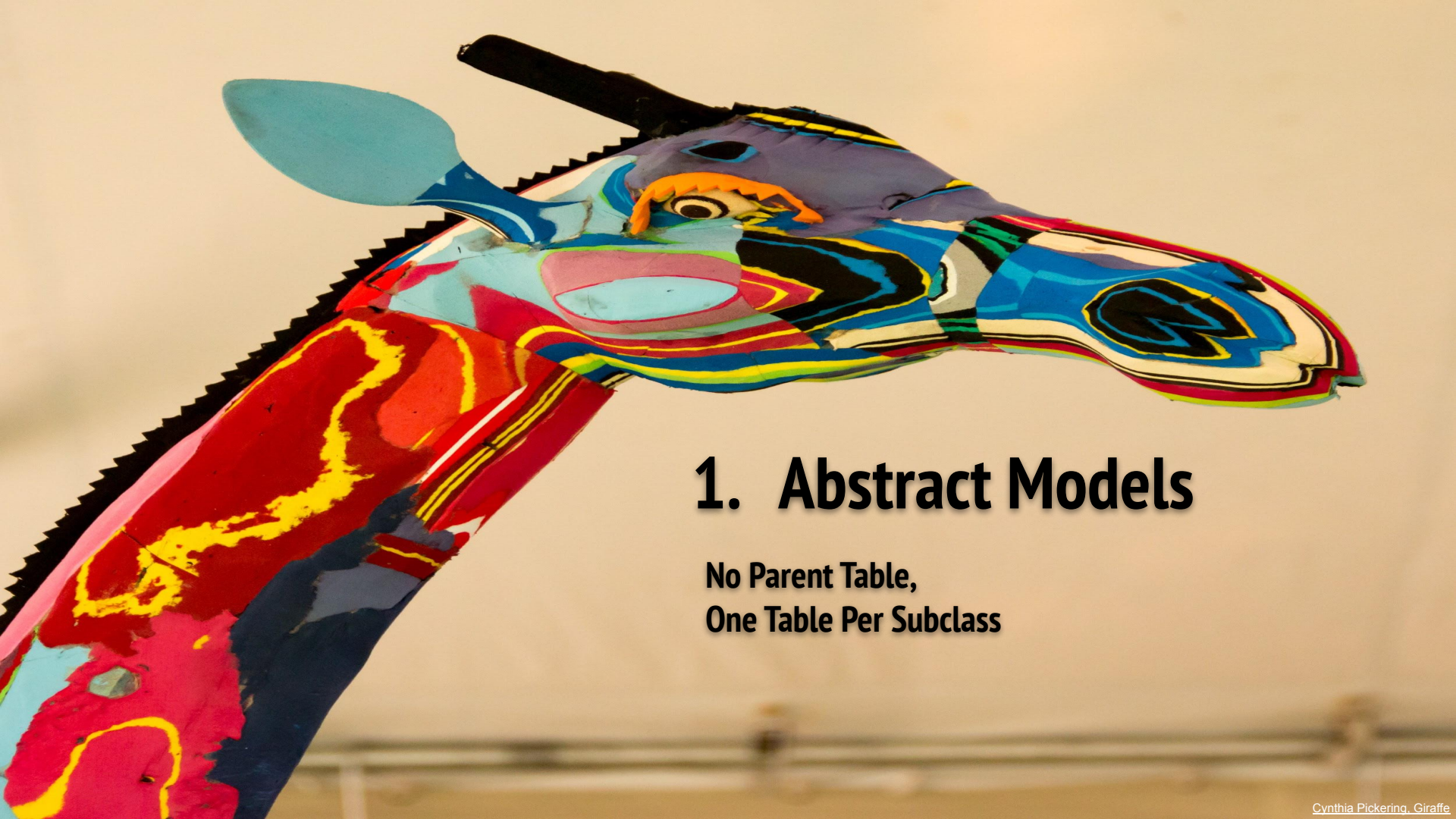
```
class Giraffe(models.Model):
    name = models.CharField(max_length=255)


class Tongue(models.Model):
    color = models.CharField(max_length=255)
    giraffe = models.OneToOneField(
        Giraffe,
        on_delete=models.CASCADE,
    )
```

Composition: OO UML vs DB Schema ERD



3 Types of Model Inheritance

Awkward!!!!



1. Abstract Models

No Parent Table,
One Table Per Subclass

Cynthia Pickering, Giraffe

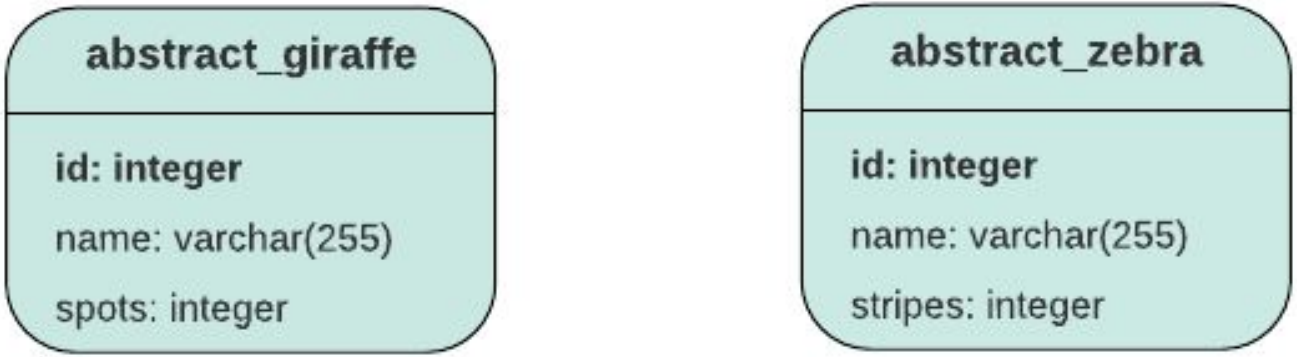
Abstract Models: Base Definition

```
class Animal(models.Model):
    class Meta:
        abstract = True
    name = models.CharField(max_length=255)

    def speak(self):
        return "##?!"
```

Abstract Models: Derived Class Definition

```
class Giraffe(Animal):
    spots = models.IntegerField()

    def speak(self):
        return "Hum"  # infrasonic
```

Abstract Models: One Table Per Subclass



Abstract Models: Query the SubClasses' table

```
In [1]: Giraffe.objects.first().speak()
Out[1]: 'hum'
```

```
SELECT
"abstract_giraffe"."id",
"abstract_giraffe"."name",
"abstract_giraffe"."spots"
FROM "abstract_giraffe"
ORDER BY "abstract_giraffe"."id" ASC
LIMIT 1
```

Use Cases: Require Multiple Duplicated Fields

- `BaseModel` that is subclassed by many models
- `TimeStampedModel` overrides `save` to update the `added` and `modified` datetime fields

Abstract Models: Ins and Outs

Ins

- Reuses the parent class's fields and field related logic
- No performance overhead in querying one table

Outs

- Parent class can not be used by itself
- Related models need multiple nullable foreign keys (Ex. `zebra_id` and `giraffe_id` are needed instead of a single `animal_id`)

2. Multi-Table Inheritance

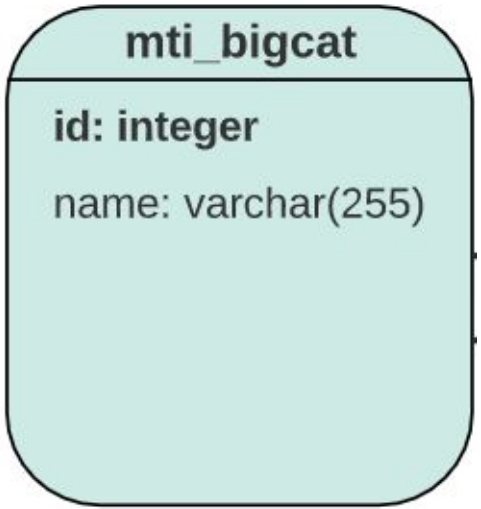
(Concrete Inheritance)

Atif Saeed,
The Ghost and the Darkness,
500px.com/matif

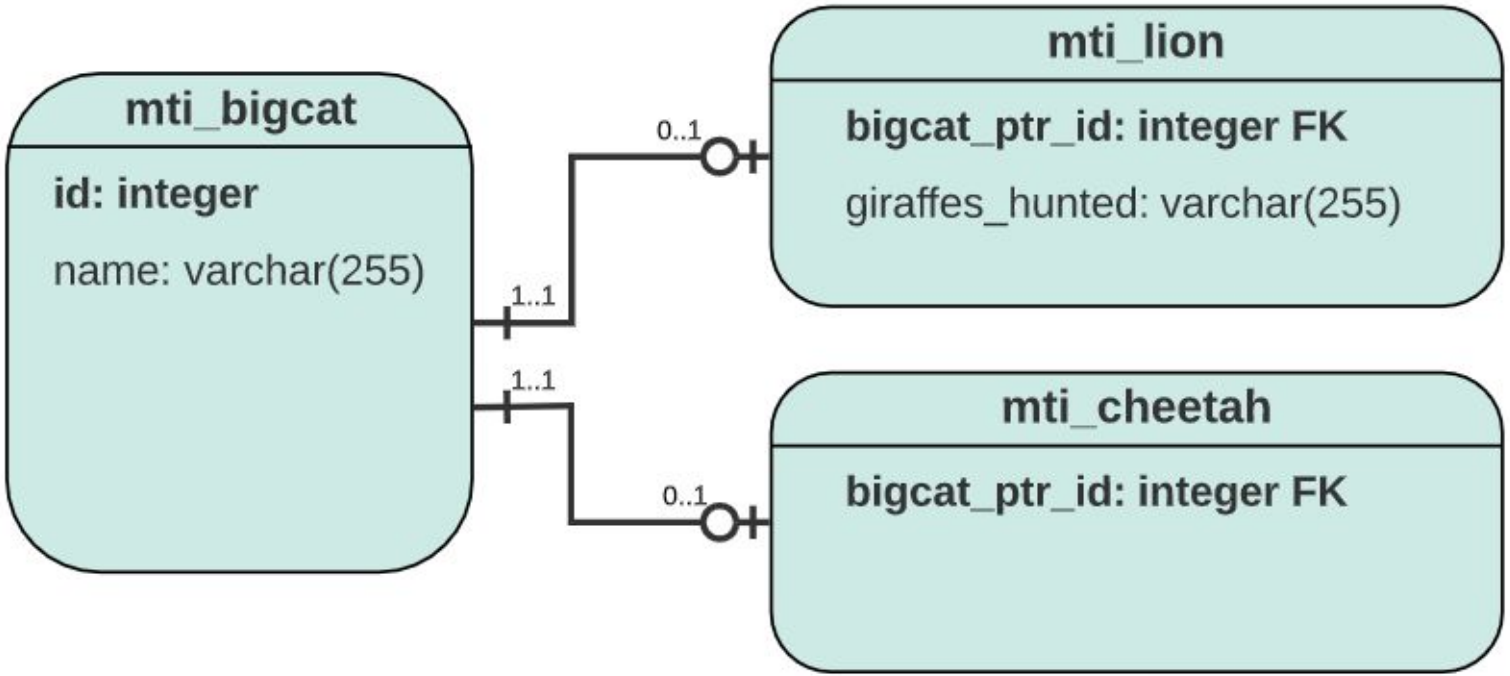
MTI: No Meta Class Definition Needed

```
class BigCat(models.Model):
    name = models.CharField(max_length=255)


class Lion(BigCat):
    giraffes_hunted = models.IntegerField()

    def speak(self):
        return "roar"
```

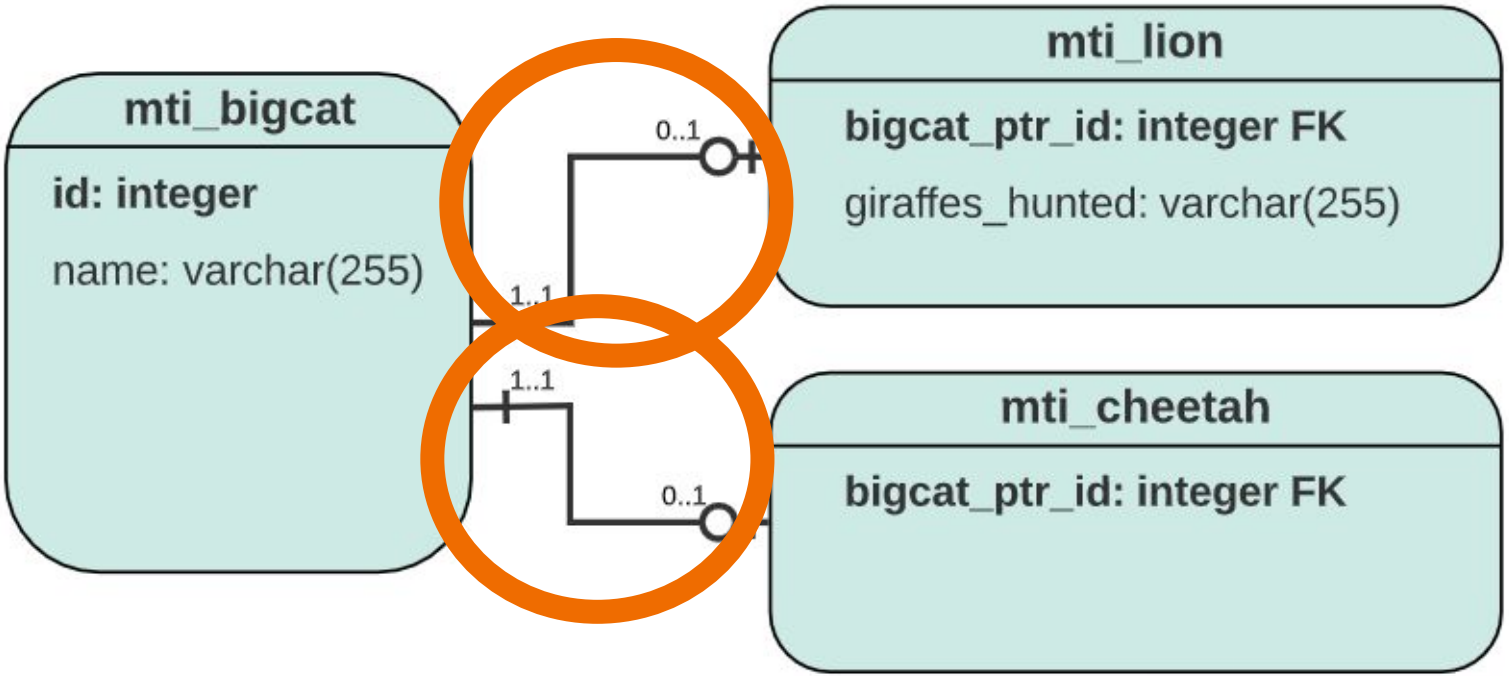
MTI: Parent is concrete



MTI: Parent is concrete; children use parent ptr as PK



MTI: You could implement this explicitly with 1:1s



MTI: Child is Queried with the Parent

```
In [1]: lions = list(Lion.objects.all())
SELECT
"mti_bigcat"."id",
"mti_bigcat"."name",
"mti_lion"."bigcat_ptr_id",
"mti_lion"."giraffes_hunted"
FROM "mti_lion"
INNER JOIN "mti_bigcat" ON (
"mti_lion"."bigcat_ptr_id" = "mti_bigcat"."id"
)
```

MTI: Accessing the Parent Requires No Queries

```
In [2]: lions[0].bigcat_ptr  # no Query, parent instance
Out[2]: <BigCat: Simba (1)>

In [3]: lions[0].name        # no Query, parent fields aliased
Out[3]: 'Simba'

In [4]: lions[0].speak()
Out[4]: 'roar'
```

MTI: Parent Query Starts Out Simple

```
In [5]: bigcats = list(BigCat.objects.all())
```

```
SELECT
"mti_bigcat"."id",
"mti_bigcat"."name"
FROM "mti_bigcat"
```

MTI: Accessing each child executes a query

```
In [6]: bigcats[0].cheetah.speak()
```

```
SELECT
"mti_bigcat"."id",
"mti_bigcat"."name",
"mti_cheetah"."bigcat_ptr_id",
FROM "mti_cheetah"
INNER JOIN "mti_bigcat"
  ON ("mti_cheetah"."bigcat_ptr_id" = "mti_bigcat"."id")
WHERE "mti_cheetah"."bigcat_ptr_id" = 1
```

RelatedObjectDoesNotExist: BigCat has no cheetah.

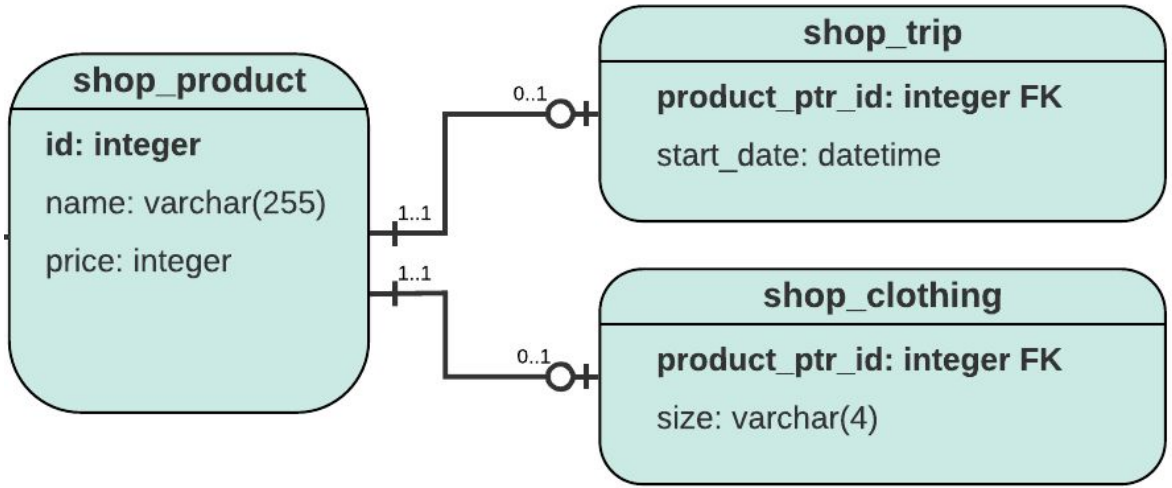
MTI: A query is needed for each subclass

```
In [7]: bigcats[0].lion.speak()
Out[7]: 'roar'
```

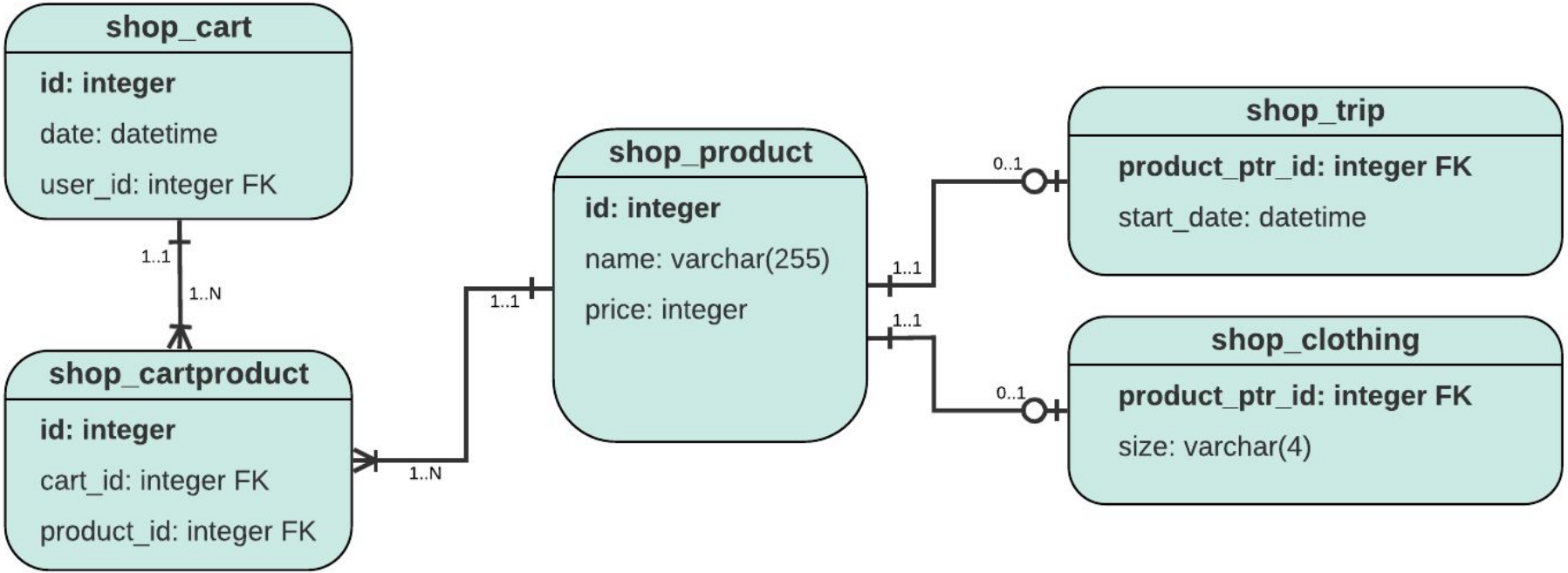
```
SELECT
"mti_bigcat"."id",
"mti_bigcat"."name",
"mti_lion"."bigcat_ptr_id",
"mti_lion"."giraffes_hunted"
FROM "mti_lion"
INNER JOIN "mti_bigcat" ON
...
```


Eager Load Optimization
1 prefetched query or LEFT JOIN per Subclass
Bernard Sprag, Solid Links

MTI Use Case: Shopping Cart



MTI Use Case: Shopping Cart



MTI Ins:

Common parent attributes can be queried together

MTI: Outs

Inefficient queries **hurt performance**

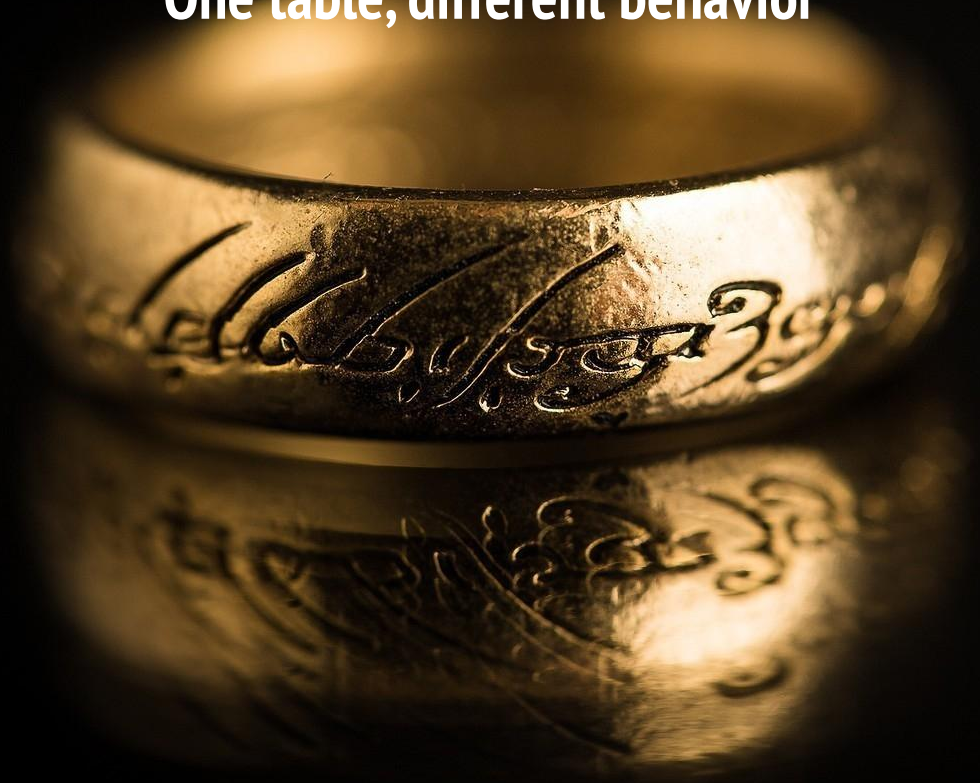
Scaling is difficult

Lack of understanding

Atif Saeed,
The Ghost and the Darkness,
500px.com/matif

3. Proxy Models:

One table, different behavior



Proxy Models: Parent Class Definition

```
class Person(models.Model):
    name = models.CharField(max_length=255)
    person_type = models.CharField(max_length=3)

    def characteristic(self):
        return "Middle Earth Dweller"
```

Proxy Models: Subclass Meta Class Sets Proxy

```
class Hobbit(Person):
    class Meta
        proxy = True

    def characteristic(self):
        return "Hairy Feet"
```

Proxy Models: 1 Table

proxy_person
id: integer
name: varchar(255)
person_type: varchar(3)

Dushan Hanuska, My Precious

Proxy Models: same data, different behavior

```
In [1]: Hobbit.objects.get(name='Frodo').characteristic()
Out[1]: 'Hairy Feet'

In [2]: Person.objects.get(name='Frodo').characteristic()
Out[2]: 'Middle Earth Dweller'
```

Proxy Models: Subclass with Custom Manager

```
class Elf(Person):
    class Meta:
        proxy = True

    objects = ElfManager()

    def characteristic(self):
        return "Immortal"
```

Proxy tables: Custom Manager Filters Elves

```
class ElfManager(models.Manager):
    def get_queryset(self):
        return super(ElfManager, self).get_queryset(
        ).filter(person_type='e')

    def create(self, **kwargs):
        kwargs.update({'person_type': 'e'})
        return super(ElfManager, self).create(**kwargs)
```

Proxy Models with Custom Managers

```
In [1]: Person.objects.all()
Out[1]: <QuerySet [<Person: Frodo (1)>, <Person: Legolas (2)>]>

In [2]: Elf.objects.all()
Out[2]: <QuerySet [<Elf: Legolas (2)>]>
```

Proxy: Same table, different filters and sorting

```
In: [1] Elf.objects.all()

SELECT
"proxy_person"."id",
"proxy_person"."name"
"proxy_person"."person_type"
FROM   "proxy_person"
WHERE  "proxy_person"."person_type" = 'e'
```

Proxy Models: Ins and Outs

Ins: Easy to modify the behavior of the subclasses

Outs: Fields used by any subclasses must be defined for all

Use Cases:

- `OrderedModel` to sort on a field
- `ActiveModel` to filter `active=True`
- `CustomUserModel` for special `User` behavior (but it might be better to create a `UserProfile` 1:1 relationship instead)

Downcasting and Single Table Inheritance

Querying with Person Returns Person Instances

```
In [1]: people = Person.objects.all()

In [2]: people[0]
Out[2]: <Person: Frodo (1)>

In [3]: people[0].characteristic()
Out[3]: "Middle Earth Dweller"
```

With “Downcasting” it returns the subclass instances

```
In [1]: Person.objects.all()
Out[1]: <QuerySet [<Hobbit: Frodo (1)>, <Elf: Legolas (2)>]>

In [2]: frodo = Person.objects.get(name="Frodo")

In [3]: frodo.characteristic()
Out[3]: "Hairy Feet"
```

Downcasting Packages and Implementation

Single-Table Inheritance

- django-typed-models
- Django STI on the Cheap

Multi-Table Inheritance (CAREFUL!)

- django-polymorphic Prefetches 1 query per derived class
- django-model-utils Adds 1 join (select_related) per derived class

Single Table Inheritance Ins and Outs

Ins

- Performance: One table means one query
- Related models need only one foreign key (`person_id`)

Outs

- Clutter and Bloat: custom subclass fields are represented with nullable or defaulted columns

STI and MTI have Similar Use Cases

Single Table Inheritance

- Subclassed models **share most fields** and relationships
- Need to query across all subclasses

Multi Table Inheritance

- Subclassed models **have many different fields**
- There is a ton of data (manually shard it by using multiple tables)

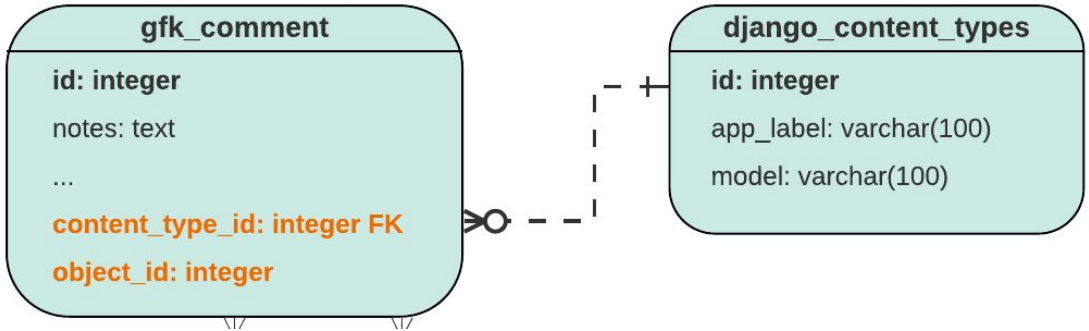
The Best Type of Model Inheritance is No Inheritance

2 Alternative Approaches and
Rethinking the Problem

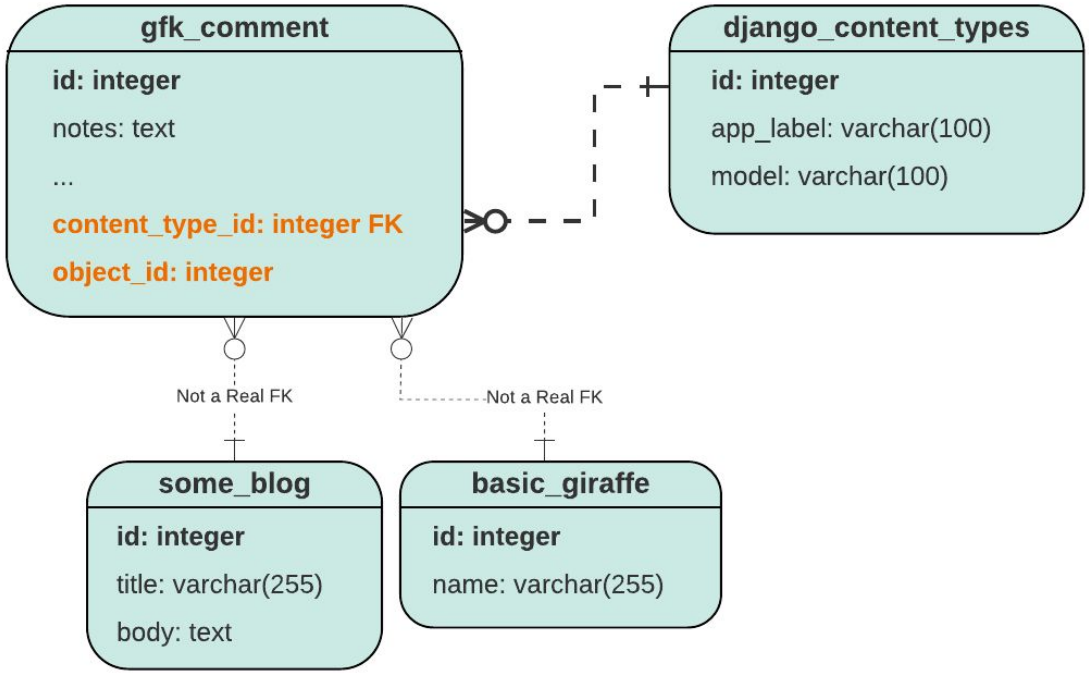
1. Polymorphism with Generic Foreign Keys

Use Caution!

Generic Foreign Keys: ERD Diagram



Generic Foreign Keys: ERD Diagram



Generic Foreign Keys: Ins and Outs

Use Cases: Tags, Comments, Likes, Favorites

Ins: **Any model can be used** and it does not need a migration

Outs:

- Code can become **hard to maintain**
  - dynamic type checking means the object could be anything
    select_related is disallowed; custom sql needed for optimization
- **No DB referential integrity** could lead to dirty data
- **Performance hit** trying to query through to related objects

2. Unstructured Data (JSONField or json-fields Package)

Use Cases: Metadata, Events, Logs, Analytics

Ins:

- Avoids the clutter of nullable fields (vs STI)
- No need for related objects (vs MTI)

Outs:

- Tough to query against unstructured fields (except with Postgres)
- Validation and data integrity are not enforced by the database

Blythe Dunham, Skier ponders Corbet's

Maybe we can rethink this

Most is-a relationships can be expressed as has-a

Inheritance	Composition
A `User` is a `Seller`	A `User` has a `SellerProfile`
A `Lion` is a `Carnivore`	A `Lion` has a `CarnivorousDiet`
A `Manager` is an `Employee`	An `Employee` has a `ManagerialJob`

Be Explicit: Your Future Self Thanks You

- Use Multiple foreign keys instead of inheritance
- Repeat yourself (a little bit) instead of **Abstract Models**
- **Implement Multi Table Inheritance explicitly** with foreign keys (ForeignKey and OneToOneField)

Thank You!

http://github.com/blythedunham/dmi

http://linkedin/in/blythedunham

blythe@snowgiraffe.com

Blythe Dunham, Powpow for Breakfast