First refer this : https://www.javatpoint.com/django-orm-queries

Simple SQL query is faster than orm .

Lets see how can we work on python manage.py shell

First migrate all the tables on terminal then type python manage.py shell on terminal.

Then type from myapp.models.import\* (to import all the models)[replace myapp name with your app name]

How can I add values to a model .

Mymodel\_intance=mymodel(field1=value1, field2=value2)

Mymodel\_intance.save()

Other method:

.mymodel.objects.create(field1=value1, field2=value2)

Update a field.

# Update the value of a field using the update() method

MyModel.objects.filter(id=1).update(myfield='newvalue')

# Query the model instance you want to update

myobject = MyModel.objects.get(id=1)

# Update the value of a field by modifying the model instance

myobject.myfield = 'newvalue'

myobject.save()

Delete a field:

# Query the object you want to delete

myobject = MyModel.objects.get(id=1)

# Delete the object using the delete() method

myobject.delete()

Types of orm: (Alternatives of django orm)

SQLAlchemy,NHibernate, Hibernnate,Jooq ,Dapper,Sql object,

Apache OpenJPA

Here are some languages and frameworks that use ORM:

ORM is not only the feature of python django.

Java: Java has several ORM frameworks, including Hibernate, Java Persistence API (JPA), MyBatis, and Spring Data JPA.

Ruby: Ruby has several ORM frameworks, including ActiveRecord (which is part of Ruby on Rails), DataMapper, and Sequel.

Python: Besides Django ORM, Python has several ORM frameworks, including SQLAlchemy, Peewee, Pony ORM, and Tortoise ORM.

PHP: PHP has several ORM frameworks, including Doctrine, Propel, and Laravel Eloquent ORM.

.NET: The .NET Framework has several ORM frameworks, including Entity Framework, NHibernate, and Dapper.

ORM frameworks are popular because they provide a high-level interface for working with databases, allowing developers to focus on application logic instead of database details. They can help to reduce the amount of code required to interact with databases, improve performance, and increase code reusability.

While there is only one ORM in Django(Django orm), there are other third-party ORMs available for Python, such as SQLAlchemy, PEEWEE, Tortoise ORM and Pony ORM. However, the Django ORM is considered one of the most popular and widely used ORMs in the Python ecosystem due to its simplicity and ease of use.

The choice of ORM library depends on the specific needs of the project, such as the type of database, performance requirements, and development team experience. Django's built-in ORM is a powerful and easy-to-use option for working with databases in Django projects, but developers can choose to use other libraries if they prefer.

ORM - (is a design or architectural pattern)

->It manges database connectivity.

->No SQL (uses primary language).

->Generates SQL code.

->Manages serialization / data types.

->code maintenance.

->security.

->one language(python).

Orm building blocks.

->tables/fields.

->Managers

->Queryset.

->Backend(DB systems).

Eg: Student[model] .objects[Manager] .all()[Query set API]

student: This is the name of the model class. In Django, models represent database tables and the attributes of those tables.

objects: This is a manager object that allows you to perform database queries on the model. The default manager is named objects and is created automatically by Django, but you can define your own custom managers as well.

all(): This is a method of the manager object that returns all the objects in the database table represented by the model. It generates a SELECT query that retrieves all the rows from the table and returns them as a QuerySet object. A QuerySet is a list-like object that allows you to manipulate and filter the results of the query.

Query set is a collection of objects from the DB.

ORM: object relational mapping

Data types:

In ORM (Object-Relational Mapping), datatypes are used to map between the types of data used in the programming language and the data types used in the database.

1.Integer: This datatype is used for storing whole numbers.

2.String: This datatype is used for storing text.

3.Boolean: This datatype is used for storing true/false values.

4.Float: This datatype is used for storing decimal numbers.

5.Date: This datatype is used for storing dates.

6.Time: This datatype is used for storing times.

7.DateTime: This datatype is used for storing both date and time.

8.Binary: This datatype is used for storing binary data such as images or files.

9.Decimal: used for storing decimal numbers with a fixed number of digits.

10. PositiveInteger :  is used to store positive integers in a database table.

11.BigInteger :to store large integers in a database table.

In addition to these common data types, some ORMs may also provide support for more advanced data types, such as JSON or array types, that are specific to certain databases or programming languages.

Note: [ URL = models.SlugField() . The purpose of using a slug field is to create a more readable, user-friendly URL that is easy to remember, share, and search for. For example, instead of using a URL like "example.com/posts/123", a slug field can be used to generate a URL like "example.com/posts/my-first-blog-post".

In an ORM, a slug field is usually defined as a string field with a maximum length, and it is generated automatically based on the value of another field, such as a title or a name, using a function that converts the string to a slug. The slug field can then be used in a query to retrieve a specific record from the database, or as part of a URL to display a web page. ]

one-to-one relationship in Django ORM.

A one-to-one relationship is a type of relationship between two models where each instance of one model is associated with exactly one instance of another model, and vice versa. In other words, each instance of the first model has a unique and corresponding instance of the second model.

Let's say we have two models: Person and Profile. Each Person can have only one Profile associated with them, and each Profile can be associated with only one Person. This is a one-to-one relationship.

from django.db import models

class Person(models.Model):

    name = models.CharField(max\_length=100)

    # other fields…

class Profile(models.Model):

    person = models.OneToOneField(Person, on\_delete=models.CASCADE, related\_name='profile')

    bio = models.TextField()

    # other fields…

In this example, we define a one-to-one relationship between Person and Profile using the OneToOneField field. The on\_delete parameter specifies what should happen when a Person object is deleted (in this case, we want to delete the associated Profile object as well). The related\_name parameter specifies the name of the reverse relation from Profile to Person.

Now, let's say we want to create a Person object with a Profile object associated with it:

person = Person.objects.create(name='John')

profile = Profile.objects.create(person=person, bio='Some bio information')

Here, we create a Person object with the name 'John' and a Profile object with the associated Person object and some bio information. We can access the Profile object associated with a Person object like this:

person = Person.objects.get(name='John')

profile = person.profile

print(profile.bio)

person = Person.objects.select\_related('profile').get(name='John')

print(person.profile.bio)

This will return the Profile object associated with the Person object. We can also access the Person object associated with a Profile object like this:

profile = Profile.objects.get(bio='Some bio information')

Person=profile.person

Print(person.name)

profile = Profile.objects.select\_related('person').get(bio='Some bio information')

person = profile.person

print(person.name)

This will return the Person object associated with the Profile object.

many-to-one relation

A many-to-one relation in Django ORM is typically defined using a ForeignKey field. Here's an example of a simple many-to-one relationship between two models in Django ORM:

from django.db import models

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=100)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

In this example, each book has exactly one author, but each author can have many books. The ForeignKey field in the Book model creates a many-to-one relationship between Book and Author. When a Book object is created, it is required to have a corresponding Author object. The on\_delete=models.CASCADE parameter specifies that if an Author object is deleted, all related Book objects will also be deleted.

You can access the author of a book using the author attribute, for example:

book = Book.objects.first()

author = book.author

Print(author.name)

This will give you the Author object associated with the first Book object in the database. You can also access all books by a given author using the book\_set attribute, for example:

author = Author.objects.first()

books = author.book\_set.all()

For book in books:

Print(book.title)

author = Author.objects.first()

books = Book.objects.filter(author\_\_name=author.name)

for book in books:

print(book.title)

author = Author.objects.first()

books = Book.objects.filter(author\_\_name=author.name).select\_related('author')

# Access the related Author object for each Book object without making a separate database query

for book in books:

author\_name = book.author.name

print(f"{book.title} (written by {author\_name})")

This will give you a QuerySet of all Book objects associated with the first Author object in the database.

how many book written by an author

author = Author.objects.select\_related('book\_set').get(name='Author Name')

book\_count = author.book\_set.count()

print(book\_count)

Many to many

In Django ORM, a many-to-many relationship is a relationship between two models where each instance of one model can be associated with multiple instances of another model, and vice versa. In this type of relationship, a join table is created to connect the two models.

Here's an example to illustrate this concept:

Suppose you have two models, Student and Course, and each student can enroll in multiple courses, and each course can have multiple students. You can create a many-to-many relationship between these two models using the ManyToManyField field.

from django.db import models

class Student(models.Model):

name = models.CharField(max\_length=100)

    def \_\_str\_\_(self):

        return self.name

class Course(models.Model):

    name = models.CharField(max\_length=100)

    students = models.ManyToManyField('Student')

    def \_\_str\_\_(self):

        return self.name

In the above example, the students field in the Course model defines a many-to-many relationship with the Student model. Django automatically creates a join table named course\_students to connect the two models.

You can use this relationship to query the related objects, for example:

# Create some students

s1 = Student.objects.create(name='Alice')

s2 = Student.objects.create(name='Bob')

s3 = Student.objects.create(name='Charlie')

# Create some courses

c1 = Course.objects.create(name='Math')

c2 = Course.objects.create(name='Science')

# Enroll students in courses

c1.students.add(s1, s2)

c2.students.add(s2, s3)

# Query courses for a specific student

s = Student.objects.get(name='Bob')

courses = s.course\_set.all()

print(courses)

# Output: <QuerySet [<Course: Math>, <Course: Science>]>

# Query students for a specific course

c = Course.objects.get(name='Math')

students = c.students.all()

print(students)

# Output: <QuerySet [<Student: Alice>, <Student: Bob>]>

In the above example, we create some students and courses, and then enroll students in courses using the add() method provided by the many-to-many relationship. We can then query courses for a specific student or students for a specific course using the related manager provided by Django (course\_set or students).

Is there any difference in manytomany relation and foreignkey in orm query

Yes, there is a difference between Foreign Key and ManyToManyField in Django's ORM queries.

ForeignKey is a relationship ( one-to-many ) between two models where one model has a reference to the other model through a foreign key. This creates a one-to-many relationship, where one instance of a model can be associated with multiple instances of another model, but each instance of the other model can only be associated with one instance of the first model. In an ORM query, you can use select\_related to retrieve related objects and prefetch\_related to retrieve multiple related objects efficiently.

ManyToManyField is a relationship between two models where each instance of one model can be associated with multiple instances of another model, and vice versa. In other words, it creates a many-to-many relationship. In an ORM query, you can use prefetch\_related to retrieve multiple related objects efficiently.

To illustrate the difference, consider the following example:

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=100)

authors = models.Foreignkey(Author)

In this example, each book can have multiple authors, and each author can be associated with multiple books. To retrieve all the books by a given author using a ForeignKey, you would use a query like this:

books = Book.objects.filter(authors\_\_name='John')

# Loop over the books and print their titles

for book in books:

print(book.title)

books = Book.objects.select\_related('authors').filter(authors\_\_name='John')

for book in books:

print(book.title)

book = Book.objects.get(title='War and Peace')

author\_name = book.authors.name

print(author\_name)

book = Book.objects.select\_related('authors').get(title='War and Peace')

author\_name = book.authors.name

print(author\_name)

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=100)

authors = models.ManyToManyField(Author)

To retrieve all the authors for a given book using a ManyToManyField, you would use a query like this:

author = Author.objects.get(name='John')

books = author.book\_set.all()

for book in books:

print(book.title)

author = Author.objects.prefetch\_related('book\_set').get(name='John')

for book in author.book\_set.all():

print(book.title)

book = Book.objects.get(title='War and Peace')

authors = book.authors.all()

for author in authors:

print(author.name)

book = Book.objects.prefetch\_related('authors').get(title='War and Peace')

authors = book.authors.all()

for author in authors:

print(author.name)

In summary, ForeignKey is used to create a one-to-many relationship, and ManyToManyField is used to create a many-to-many relationship. The choice between the two depends on the specific needs of your application.

Example for union ,intersection and difference

Here are some examples of using the Django ORM for union, intersection, and difference operations on querysets:

Assuming we have two models: Book and Author, where Book has a foreign key to Author:

from django.db.models import Q

class Author(models.Model):

name = models.CharField(max\_length=255)

# other fields…

class Book(models.Model):

title = models.CharField(max\_length=255)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

# other fields…

1.Union:

To perform a union operation on two querysets, you can use the ’ |’ (OR) operator or the union() method. For example, to get all books written by either "John" or "Jane":

john\_books = Book.objects.filter(author\_\_name='John')

jane\_books = Book.objects.filter(author\_\_name='Jane')

# using the | operator:

result = john\_books | jane\_books

# or using the union() method:

result = john\_books.union(jane\_books)

print("Books by John or Jane:")

for book in result:

print(book.title)

2.Intersection:

To perform an intersection operation on two querysets, you can use the & (AND) operator or the intersection() method. For example, to get all books written by both "John" and "Jane":

john\_books = Book.objects.filter(author\_\_name='John')

jane\_books = Book.objects.filter(author\_\_name='Jane')

# using the & operator:

result = john\_books & jane\_books

# or using the intersection() method:

result = john\_books.intersection(jane\_books)

Note that the intersection of two querysets can be an empty queryset if there are no common items between them.

3.Difference:

To perform a difference operation on two querysets, you can use the - operator or the difference() method. For example, to get all books written by "John" but not "Jane":

john\_books = Book.objects.filter(author\_\_name='John')

jane\_books = Book.objects.filter(author\_\_name='Jane')

# using the - operator:

result = john\_books - jane\_books

# or using the difference() method:

result = john\_books.difference(jane\_books)

Note that the difference operation is not commutative, so john\_books - jane\_books is not the same as jane\_books - john\_books.

Here are some notes about various Django ORM concepts:

1. Get: Retrieve a single object by its primary key

The get() method is used to retrieve a single object that matches the specified lookup parameters.

from django.db import models

class MyModel(models.Model):

id = models.AutoField(primary\_key=True)

from myapp.models import MyModel

# Retrieve an object by its primary key

obj = MyModel.objects.get(pk=1)

2.Filter: Retrieve a set of objects that match a specific lookup parameter

The filter() method is used to retrieve a set of objects that match the specified lookup parameters.

from myapp.models import MyModel

# Retrieve all objects where the 'name' field contains 'John'

objs = MyModel.objects.filter(name\_\_contains='John')

3.Q object: Build a complex query using OR conditions.

The Q object is used to build complex queries that involve OR conditions between different lookup parameters.

from django.db.models import Q

from myapp.models import MyModel

# Retrieve all objects where the 'name' field contains 'John' OR the 'age' field is greater than 30

objs = MyModel.objects.filter(Q(name\_\_contains='John') | Q(age\_\_gt=30))

1. Annotate vs Aggregate:

Annotate vs Aggregate: annotate() is used to add a calculated field to each object in a queryset, while aggregate() is used to calculate a single value for the entire queryset.

from django.db import models

class MyModel(models.Model):

children = models.ManyToManyField('self')

age = models.IntegerField()

# Add a calculated field to each object in the queryset

objs = MyModel.objects.annotate(num\_children=Count('children'))

for obj in objs:

print(f"Object {obj.id} has {obj.num\_children} children.")

# Calculate the average age of all objects in the queryset

avg\_age = MyModel.objects.aggregate(Avg('age'))

print("The average age is:", avg\_age['age\_\_avg'])

5.Values and Values list: Retrieve specific fields from a queryset

from myapp.models import MyModel

# Retrieve a list of dictionaries containing only the 'name' and 'age' fields

values = MyModel.objects.values('name', 'age')

for value in values:

print(value)

values = MyModel.objects.values('name', 'age')

for value in values:

print(value['name'], value['age'])

# Retrieve a list of tuples containing only the 'name' and 'age' fields

values\_list = MyModel.objects.values\_list('name', 'age')

for value in values\_list:

print(value)

values\_list = MyModel.objects.values\_list('name', 'age')

for value in values\_list:

print(value[0], value[1])

1. F and FO object: Refer to a field on the model, or apply a database function to a field.

Eg:

from django.db.models import F, Func

# Retrieve all objects where the value of 'field1' is greater than the value of 'field2'

queryset = MyModel.objects.filter(field1\_\_gt=F('field2'))

# Retrieve all objects and convert the 'date\_field' to a string using the 'TO\_CHAR' function

queryset = MyModel.objects.annotate(date\_string=Func(F('date\_field'), function='TO\_CHAR'))

for obj in queryset:

print(obj.date\_string)

from django.db.models import F, Func

from django.db import models

from django.db.models.functions import Sqrt

class MyModel(models.Model):

age = models.IntegerField()

score = models.FloatField()

# Retrieve all objects where the 'age' field is greater than the 'score' field

objs = MyModel.objects.filter(age\_\_gt=F('score'))

# Retrieve all objects where the square root of the 'score' field is greater than 10

objs = MyModel.objects.filter(Func(F('score'), function=Sqrt()) > 10)

The main purpose of the F object is to allow you to make database queries that involve comparing or updating values of fields in the database without having to fetch the values into Python memory first. This can save time and resources, particularly when dealing with large datasets.

from django.db.models import F

Book.objects.filter(price\_\_gt=10).update(price=F('price') \* 0.9)

In this example, the F object is used to refer to the current value of the price field in the database, so that you can multiply it by 0.9 and update the value in the database without having to fetch the values into Python memory first.

Overall, the F object can be a powerful tool for optimizing database queries in Django, particularly when dealing with large datasets.

7.Managers: Define custom queryset methods.

manager is a class that defines methods for querying the database for a particular model.

Default managers have methods like **all(), create(), get(), filter(), update(), and delete()** that allow you to query and modify the database.. The primary use of managers is to provide custom query methods for a model.

The code defines a custom manager MyModelManager for a Django model MyModel. The manager provides a custom queryset MyModelQuerySet and overrides the get\_queryset method to use it. It also defines a custom method custom\_method that operates on the queryset.

Note that it's important to use the self.model attribute when creating the queryset, rather than hard-coding the model class, so that the manager works correctly with subclasses.

We can also define custom managers for your models by creating a new class that inherits from **models.Manager** and adding it as an attribute to your model. Custom managers allow you to define new methods for querying the database that are specific to your application's needs.

For example, let's say you have a **Book** model that has a field called **published\_date**. You could create a custom manager that returns all books that were published in a certain year like this:

from django.db import models

class BookManager(models.Manager):

def published\_in\_year(self, year):

return self.filter(published\_date\_\_year=year)

class Book(models.Model):

title = models.CharField(max\_length=255)

published\_date = models.DateField()

objects = models.Manager() # Default manager

published\_books = BookManager() # Custom manager

Now you can use the **published\_books** manager to query for books that were published in a specific year like this:

from myapp.models import Book

books\_in\_2022 = Book.published\_books.published\_in\_year(2022)

In this example, the **published\_in\_year** method is defined on the **BookManager** class and returns a queryset that filters books by the **published\_date** field. This method can then be accessed on the **published\_books** manager that we defined for the **Book** model.

1. Types of model inheritance in Django: Abstract base classes (ABC), multi-table inheritance, and proxy models.

model inheritance allows you to define a new model based on an existing model, which inherits all the fields and methods of the existing model. This can help you avoid duplicating code and make it easier to maintain your codebase.

Abstract base classes: An abstract base class is a model that is not meant to be instantiated on its own. Instead, it provides a set of common fields and methods that can be inherited by other models. To define an abstract base class, you can set the abstract attribute to True in the model's Meta class.

from django.db import models

# Abstract base class

class MyBaseModel(models.Model):

name = models.CharField(max\_length=255)

    class Meta:

        abstract = True

Multi-table inheritance: Multi-table inheritance creates a new table for each model in the inheritance chain. Each table contains the fields of the corresponding model, as well as the fields inherited from the parent models. To define a model that uses multi-table inheritance, you can define a foreign key to the parent model in the child model.

# Multi-table inheritance

class MyChildModel(MyBaseModel):

    age = models.IntegerField()

Proxy models: A proxy model is a model that provides a different interface to the same data as an existing model. Proxy models can be used to add new methods or attributes to an existing model, or to change the default ordering or filtering of the model. To define a proxy model, you can set the proxy attribute to True in the model's Meta class.

# Proxy model

class MyProxyModel(MyBaseModel):

    class Meta:

        proxy = True

Use of proxy in table inheritence in djago orm

In Django's ORM (Object-Relational Mapping), proxy models can be used in table inheritance to provide different behaviors or additional functionality for existing models without changing the underlying database schema. When using table inheritance, a proxy model refers to a model that shares the same database table with its parent model but behaves as a separate entity.

Here are some use cases for using proxy models in table inheritance in Django ORM:

Customizing model behavior: You can create a proxy model that inherits from an existing model and override certain methods, fields, or behavior to provide specialized functionality. This allows you to modify the behavior of the model without affecting the original model or its associated database table.

Customizing model representation: Proxy models can override the \_\_str\_\_ method to provide a different string representation for the model instances. This can be useful when you want to display the model instances differently in the Django admin site or other parts of your application.

Extending functionality: Proxy models can be used to add extra fields or methods to an existing model without modifying its structure. This can be useful when you want to extend the functionality of a model without impacting its existing codebase or affecting the original table schema.

Permission-based access: You can use proxy models to apply different permissions or access controls to specific subsets of data within a table. By defining proxy models that inherit from the original model and applying permissions to those proxy models, you can control access to the data based on different criteria.

To create a proxy model in Django, you can define a new model class that inherits from the original model and set the proxy attribute to True in the Meta class. This tells Django that the model is a proxy model and should share the same database table with the parent model.

Here's an example of defining a proxy model in Django:

from django.db import models

class BaseModel(models.Model):

# Common fields and methods

class Meta:

abstract = True

class OriginalModel(BaseModel):

# Fields and methods for the original model

class Meta:

db\_table = 'original\_model'

class ProxyModel(OriginalModel):

# Additional fields or overridden methods

class Meta:

proxy = True

In this example, BaseModel serves as an abstract base class containing common fields and methods shared by multiple models. OriginalModel is the main model, and ProxyModel is a proxy model inheriting from OriginalModel and providing additional functionality or customization.

Remember to register your models in the Django admin or use them in your application as needed.

Using proxy models in table inheritance allows you to create specialized versions of existing models, modify behavior, or add functionality without altering the original model or database schema.

9.Meta class attributes: Define model metadata

Meta class attributes: The Meta class is used to define metadata for the model, such as ordering, indexes, and verbose names.

from django.db import models

class MyModel(models.Model):

name = models.CharField(max\_length=255)

    class Meta:

        ordering = ['name']

        indexes = [models.Index(fields=['name'])]

        verbose\_name\_plural = 'My Models'

Meta attributes: ordering, indexes, verbose\_name, proxy, abstract.

1. Prefetch related and Select related (join):

prefetch\_related() and select\_related() are used to reduce the number of database queries needed to retrieve related objects.

Select related is known as joins in orm.

[ select\_related is used to retrieve objects that are related via a ForeignKey or OneToOneField

 prefetch\_related is more appropriate for many-to-many and reverse foreign key relations ]

Prefetch related:

"Prefetch related" allows to retrieve related objects in a separate query, after the initial query for the main object has been executed. This can be useful when you need to access related objects in a loop or when you have a one-to-many or many-to-many relationship.

For example, consider the following models:

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=100)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

published\_date = models.DateField()

class Review(models.Model):

book = models.ForeignKey(Book, on\_delete=models.CASCADE)

reviewer\_name = models.CharField(max\_length=100)

text = models.TextField()

If you want to retrieve all books and their authors, as well as all reviews for each book, you can use the following code:

books = Book.objects.prefetch\_related('author', 'review\_set')

for book in books:

print(f"Title: {book.title}")

print(f"Author: {book.author.name}")

print("Reviews:")

for review in book.review\_set.all():

print(f"{review.reviewer\_name}: {review.text}")

print("\n")

This will retrieve all books and their authors in one query, and all reviews for those books in a separate query. The related objects are stored in a cache, which allows you to access them efficiently in a loop or when iterating over the queryset.

Note that the argument 'review\_set' is used because Django automatically creates a reverse relationship from Book to Review, and the default name for this relationship is review\_set. If you have specified a different name for the reverse relationship, you should use that name instead.

Select related (join):

 "select related" is a query optimization technique in Django's ORM that allows you to retrieve related objects in a single database query instead of multiple queries. When you use select\_related, the ORM follows foreign key relationships and retrieves the related objects in the same query as the original object. This can improve performance and reduce the number of database queries needed to fetch related objects.

For example, consider the following models:

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=100)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

If you want to retrieve all books and their authors, you can use the following code:

books = Book.objects.all().select\_related('author')

for book in books:

print(f"Title: {book.title}")

print(f"Author: {book.author.name}\n")

This will retrieve all books and their authors in a single database query, instead of one query for the books and one query for each author.

Another eg:

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=200)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

Here's an example Django query that uses annotate, prefetch\_related, and select\_related to retrieve all authors and their books, along with the number of books each author has:

Authors = Author.objects.prefetch\_related('book\_set').annotate(num\_books=Count('book'))

.select\_related('book')

for author in authors:

print(author.name, author.num\_books)

for book in author.book\_set.all():

print(book.title)

* prefetch\_related('book\_set'): This tells Django to prefetch all related Book objects for each author, using the book\_set related name that Django automatically generates for reverse foreign key relationships.
* annotate(num\_books=Count('book')): This adds a num\_books field to each Author object, which contains the number of related Book objects for that author.
* select\_related('book'): This tells Django to use a SQL join to retrieve the related Book objects for each author, which can improve performance for queries that access related objects.

In Django's ORM (Object-Relational Mapping), joins are performed implicitly based on the defined relationships between models. Django automatically generates the necessary SQL JOIN statements to retrieve related data.

Here are the types of joins commonly used in Django ORM:

JOINS.

****Inner Join****: The default type of join in Django ORM. It retrieves only the records where there is a match in both tables/models being joined. It is used when you access related objects through foreign key or one-to-one relationships.

# Retrieve all books along with their corresponding authors

Book.objects.select\_related('author')

****Left Outer Join****: Retrieves all the records from the left table/model and the matching records from the right table/model. If there is no match, it returns NULL values for the right table/model. It is used when you access related objects through a one-to-many or many-to-many relationship.

# Retrieve all authors along with their books (including authors without any books)

Author.objects.prefetch\_related('book\_set')

****Explicit Joins****: In some cases, you might need to perform explicit joins to retrieve specific data from related models. This can be achieved using the **join()** method and specifying the join conditions.

from django.db.models import F

# Retrieve all books with the same author name as a given book

given\_book = Book.objects.get(id=1)

books = Book.objects.filter(author\_\_name=F('given\_book\_\_author\_\_name'))

Django's ORM abstracts away the need for explicitly writing JOIN statements in most cases. Instead, you work with models and their relationships, and Django handles the underlying JOIN operations for you.

11.Signals:

Signals are used to trigger certain actions when certain events occur in the database, such as saving or deleting a model instance.

Django provides a set of built-in signals that are triggered by certain events in the database. These signals are defined in the django.db.models.signals module and can be used to perform certain actions when the corresponding event occurs.

Here are some examples of the built-in signals in Django:

pre\_save: Sent just before an object is saved.

post\_save: Sent just after an object is saved.

pre\_delete: Sent just before an object is deleted.

post\_delete: Sent just after an object is deleted.

Suppose we want to send an email to the user whenever a new user is created in our application.

from django.core.mail import send\_mail

from django.db.models.signals import post\_save

from django.dispatch import receiver

from django.contrib.auth.models import User

@receiver(post\_save, sender=User)

def send\_welcome\_email(sender, instance, created, \*\*kwargs):

if created:

send\_mail(

'Welcome to our application',

'Dear {},\n\nWelcome to our application!'.format(instance.username),

'noreply@example.com',

[instance.email],

fail\_silently=False,

)

12.Raw method:

The raw() method is used to execute raw SQL queries and return the results as model instances. Here is an example:

Suppose we want to retrieve all users whose age is greater than 18 using a raw SQL query.

from django.db import models

class MyModel(models.Model):

    name = models.CharField(max\_length=100)

    # other fields…

# Define the SQL query

sql\_query = 'SELECT name FROM myapp\_mymodel'

# Execute the query using the raw() method

results = MyModel.objects.raw(sql\_query)

# Loop through the results and print the names

for result in results:

    print(result.name)

from django.db import connection

# Define the SQL query with parameter substitution

sql\_query = 'SELECT name FROM myapp\_mymodel WHERE id = %s'

# Execute the query using the raw() method and pass the parameter value

results = MyModel.objects.raw(sql\_query, [1])

# Loop through the results and print the names

for result in results:

print(result.name)

1. Cursor:

The cursor() method is used to obtain a raw database cursor for executing custom SQL queries and retrieve the results as Python objects. Here is an example:

Suppose we want to retrieve all users whose age is greater than 18 using a custom SQL query and the cursor method.

from django.db import connection

with connection.cursor() as cursor:

cursor.execute('SELECT \* FROM auth\_user WHERE age > %s', [age])

users = cursor.fetchall()

for user in users:

print(user)

The connection.cursor() method returns a cursor object that can be used to execute SQL queries directly on the database. In this code, we use the cursor object to execute the SQL query and fetch all the results using cursor.fetchall(). Then, we iterate over the results and print each row using the print() statement.

14.Redirect and Reverse lazy: redirect() is used to redirect to a different URL, while reverse\_lazy() is used to reverse URL patterns in a lazy manner.

The redirect method allows us to redirect to a different URL, while the reverse lazy method allows us to retrieve the URL for a view name lazily. Here is an example:

Suppose we want to redirect to a different URL after a user logs in.

from django.shortcuts import redirect

def login\_view(request):

# Perform login logic

return redirect('home')

def home\_view(request):

# Render home page

Pass

reverse\_lazy(): The reverse\_lazy() method can be used to retrieve the URL for the home\_view by providing its name 'home' as an argument. The reverse\_lazy() method returns a lazy object that will only be resolved to the actual URL when it is needed.

Here is an example of how to use reverse\_lazy() to retrieve the URL for the home\_view:

from django.urls import reverse\_lazy

def some\_function(request):

# Get the URL for the home view

home\_url = reverse\_lazy('home')

# …

15.Aggregate functions: Django supports a variety of aggregate functions, such as Count(), Sum(), Avg(), Min(), and Max().In Django ORM, aggregate functions are used to perform calculations on a set of values for a particular field in a queryset. Aggregate functions are performed on a group of values and return a single value as the result.

Aggregate functions are used to perform operations on groups of records. Here is an example:

Suppose we want to retrieve the average rating of all books in our application.

AVG:

from django.db.models import Avg

average\_rating = Book.objects.aggregate(Avg('rating'))

print(f"Average rating: {average\_rating['rating\_\_avg']}")

Count:

Count is used to count the number of records that match a particular query.

from myapp.models import Book

# Count the number of books in the database

num\_books = Book.objects.count()

# Count the number of books in the "Fiction" category

num\_fiction\_books = Book.objects.filter(category='Fiction').count()

# Print the results

print(f"Total number of books: {num\_books}")

print(f"Number of fiction books: {num\_fiction\_books}")

from django.db.models import Count

# Get the number of books in each category

book\_counts = Book.objects.values('category').annotate(category\_count=Count('id'))

for book\_count in book\_counts:

print(f"{book\_count['category']}: {book\_count['category\_count']}")

Sum:

Sum is used to calculate the sum of the values in a particular field for a group of records.

from myapp.models import Product

total\_price =Product.objects.aggregate(total=Sum('price'))['total']

print(total\_price)

from django.db.models import Sum

# Get the total sales for each author

author\_sales = Author.objects.values('name').annotate(total\_sales=Sum('book\_\_sales'))

for author\_sale in author\_sales:

print(f"{author\_sale['name']}: {author\_sale['total\_sales']}")

Max:

Max is used to retrieve the maximum value of a particular field for a group of records.

from myapp.models import Book

max\_price = Book.objects.all().aggregate(max\_price=Max('price'))

print(f"The maximum price of all books is {max\_price}")

from django.db.models import Max

# Get the highest-rated book in each category

top\_books = Book.objects.values('category').annotate(max\_rating=Max('rating')).order\_by('category')

for top\_book in top\_books:

    book = Book.objects.filter(category=top\_book['category'], rating=top\_book['max\_rating']).first()

print(f"{book.title} ({book.category}): {book.rating}")

Min:

Min is used to retrieve the minimum value of a particular field for a group of records.

from myapp.models import Product

# Retrieve the minimum price of a product

min\_price = Product.objects.aggregate(min\_price=Min('price'))['min\_price']

print(f"The minimum price of a product is {min\_price}")

from django.db.models import Min

# Get the earliest published book for each author

first\_books = Author.objects.values('name').annotate(first\_book=Min('book\_\_published\_date'))

for first\_book in first\_books:

    book = Book.objects.filter(author\_\_name=first\_book['name'], published\_date=first\_book['first\_book']).first()

    print(f"{book.title} ({book.author.name}): {book.published\_date}")

These are just a few examples of the aggregate functions available in Django. There are many more, including StdDev, Variance, Sum, and CountDistinct.

16.Bulk create: The bulk\_create() method is used to create multiple model instances at once for improved efficiency.

You can create multiple instances of a model at once using the bulk\_create() method. Here's an example:

from myapp.models import MyModel

objs = [

    MyModel(id=1, name='John'),

    MyModel(id=2, name='Jane'),

    MyModel(id=3, name='Bob'),

]

MyModel.objects.bulk\_create(objs)

17.exclude(): The exclude() method is used to exclude objects from a queryset based on specified lookup parameters.

You can exclude certain instances from a query using the exclude() method. Here's an example:

from myapp.models import MyModel

# exclude all instances where age is less than 18

queryset = MyModel.objects.exclude(age\_\_lt=18)

18.Month, year, weekday, day:

We can filter instances based on month, year, weekday, and day using the month, year, weekday, and day methods respectively. Here's an example:

from myapp.models import MyModel

from datetime import date

# filter instances where the date field is in May 2023

queryset = MyModel.objects.filter(date\_\_month=5, date\_\_year=2023)

# filter instances where the date field is on a Monday

queryset = MyModel.objects.filter(date\_\_weekday=0)

# filter instances where the date field is on the 5th day of the month

queryset = MyModel.objects.filter(date\_\_day=5)

19.Contains (case sensitive), icontains (case insensitive): These are lookup parameters that can be used to filter objects based on whether a specific substring is contained within a particular field.

You can filter instances based on whether a field contains a certain string using the contains and icontains methods respectively. Here's an example:

from myapp.models import MyModel

# filter instances where the name field contains 'John'

queryset = MyModel.objects.filter(name\_\_contains='John')

# filter instances where the name field contains 'john' or 'JOHN' (case-insensitive)

queryset = MyModel.objects.filter(name\_\_icontains='john')

20.Abstract user and abstract base user:

Django provides two abstract base classes for creating custom user models: AbstractUser and AbstractBaseUser. Here's an example:

from django.contrib.auth.models import AbstractUser

from django.db import models

class CustomUser(AbstractUser):

# add custom fields here

age = models.IntegerField()

def \_\_str\_\_(self):

return self.username

In the above example, CustomUser is a custom user model that inherits from AbstractUser. You can also create a custom user model that inherits from AbstractBaseUser if you need more control over the fields and methods.

Django provides two abstract base classes AbstractUser and AbstractBaseUser to create custom user models.

In your example, you have created a custom user model CustomUser that inherits from AbstractUser. You have added a custom field age to the model using the IntegerField of the models module, which is not present in the original AbstractUser model. The \_\_str\_\_() method has also been overridden to display the username instead of the default object representation.

If you need more control over the fields and methods of the custom user model, you can inherit from AbstractBaseUser. However, AbstractBaseUser requires you to implement some methods such as get\_full\_name(), get\_short\_name(), has\_perm(), and has\_module\_perms(). So, if you inherit from AbstractBaseUser, you need to implement these methods to provide authentication and authorization features to your application.

1. AND, OR, NOT

Suppose we have a model called Person with fields name, age, and city, and we want to retrieve all people who are either aged 30 or above, and live in either "New York" or "Los Angeles", but not those whose name contains the word "John".

from django.db.models import Q

Person.objects.filter(

    Q(age\_\_gte=30) & (Q(city='New York') | Q(city='Los Angeles')) & ~Q(name\_\_icontains='John')

)

In this example:

Q(age\_\_gte=30) represents the condition where age is greater than or equal to 30.

(Q(city='New York') | Q(city='Los Angeles')) represents the condition where the city is either "New York" or "Los Angeles".

~Q(name\_\_icontains='John') represents the negation of the condition where the name contains the word "John".

The & operator is used to represent AND conditions, the | operator is used to represent OR conditions, and the ~ operator is used to represent NOT conditions.

Possible values of on\_delete in django orm:

Cascading" in Django ORM refers to the behavior of automatically deleting related objects when the primary object is deleted. The possible values for cascading in Django ORM are:

CASCADE: If you set on\_delete=models.CASCADE in a ForeignKey field, when the referenced object is deleted, all objects that have a ForeignKey to that object will also be deleted. This is the default behavior if on\_delete is not specified.

PROTECT: If you set on\_delete=models.PROTECT in a ForeignKey field, when the referenced object is deleted, the operation will be blocked and an error will be raised. This is useful if you want to prevent accidental deletions of objects that are referenced by other objects.

SET\_NULL: If you set on\_delete=models.SET\_NULL in a ForeignKey field, when the referenced object is deleted, the ForeignKey field of all objects that have a ForeignKey to that object will be set to NULL. This is useful when you want to preserve the referencing objects but break the relationship with the deleted object.

SET\_DEFAULT: If you set on\_delete=models.SET\_DEFAULT in a ForeignKey field, when the referenced object is deleted, the ForeignKey field of all objects that have a ForeignKey to that object will be set to its default value. This is useful when you want to preserve the referencing objects but set the relationship to a default value.

SET(): If you set on\_delete=models.SET(<value>) in a ForeignKey field, when the referenced object is deleted, the ForeignKey field of all objects that have a ForeignKey to that object will be set to the specified value. This is useful when you want to preserve the referencing objects but set the relationship to a specific value.

Note that these values can only be used in ForeignKey fields, which represent a many-to-one relationship between two models. They cannot be used in other types of fields such as OneToOneField, ManyToManyField, or GenericForeignKey.

What happens when we use makemigration and migrate in django shell.

When you use makemigrations and migrate commands in Django shell, the behavior is the same as when you run them from the command line.

makemigrations is used to create new database migration files based on changes in your models. When we run makemigrations, Django inspects your models and compares them to the existing migrations in your project's migrations directory. If it detects any changes in your models, it generates new migration files that reflect those changes.

migrate is used to apply the database migrations to our database. When we run migrate, Django looks for the latest migration files in your project's migrations directory and applies them to your database. It uses the information in the migration files to update the database schema, create or modify tables, add or remove columns, and so on.

Difference between filter and get in django orm.

if an model is empty filter returns empty query set (<QuerySet []>) ,while the get returns error (self.model.DoesNotExist).

We can take value using get by ::

br=Brand.objects.get(brand\_id=100)

br.name

br.price

But we cannot take values by above syntax using filter (when we use above syntax in filter it show error ).

br=Brand.objects.filter(brand\_id=100)

For brand in br:

print(brand.name, brand.price)

Other method :

br=Brand.objects.filter(brand\_id=100)

br[0].name

Br[0].id

The filter method returns a QuerySet containing all the objects that match the given conditions. The conditions are specified using keyword arguments, where each argument corresponds to a field on the model. For example, MyModel.objects.filter(name='John') will return a QuerySet containing all objects from MyModel whose name field is equal to 'John'. You can also chain multiple filters together to create more complex queries.

On the other hand, the get method returns a single object that matches the given conditions. The conditions are specified using the same keyword arguments as filter. For example, MyModel.objects.get(name='John') will return a single object from MyModel whose name field is equal to 'John'. If no object is found, a MyModel.DoesNotExist exception is raised.

In summary, filter is used to retrieve multiple objects that match certain conditions, while get is used to retrieve a single object that matches certain conditions. Use filter when you want to work with a collection of objects, and use get when you want to work with a single object.

Lazy loading

Lazy loading is a technique used in Django ORM to optimize database queries and reduce unnecessary database hits. It involves loading related objects on-demand, only when they are actually needed, rather than loading them all at once.

Here's an example to demonstrate how lazy loading works in Django ORM:

Let's say we have two models, Author and Book, where an author can have multiple books. The Author model has a foreign key relationship to the Book model:

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=100)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

Now, let's say we want to retrieve all books written by a specific author. We can use the following query:

author = Author.objects.get(name='J.K. Rowling')

books = author.book\_set.all()

In this case, books is a lazy object that doesn't actually execute the query until we try to access its data. For example, we can loop through the books object to retrieve each book's title:

for book in books:

print(book.title)

This will execute a separate database query for each book in the books object, but only when the loop reaches that specific book. This way, we don't load all the related books at once, which can be inefficient and slow, especially for large datasets.

Alternatively, we can use prefetch\_related to fetch all books in a single query, which can be more efficient for some use cases.

author = Author.objects.prefetch\_related('book\_set').get(name='J.K. Rowling')

books = author.book\_set.all()

print("Books by", author.name)

for book in books:

print(book.title)

In this case, the prefetch\_related method fetches all books related to the Author object in a single query. This can be useful when we know we will need all related objects and want to minimize the number of queries sent to the database.

In summary, lazy loading is a useful technique to optimize database queries in Django ORM. It helps reduce unnecessary database hits by loading related objects on-demand, only when they are actually needed.

To enable lazy loading in Django ORM, you can use the select\_related and prefetch\_related methods when querying the database. These methods allow you to specify which related objects should be loaded immediately and which should be loaded lazily.

auto\_now\_add

In Django, the auto\_now\_add field option is used to automatically set the field value to the current time when the object is first created. This option can be used with the DateTimeField and DateField field types.

Here is an example of using auto\_now\_add with a DateTimeField:

from django.db import models

class MyModel(models.Model):

created\_at = models.DateTimeField(auto\_now\_add=True)

# other fields…

In this example, the created\_at field will be set to the current date and time when the object is first created, and it will not change when the object is updated.

The auto\_now\_add option ensures that the value of the field is set automatically and consistently, without requiring the developer to manually set the field value during object creation.

It's worth noting that if you set both auto\_now\_add and auto\_now on a single field, the auto\_now\_add option takes precedence over auto\_now. This means that the field will only be set to the current time when the object is created, and it will not be updated to the current time when the object is saved.

Lookup parameters.

In Django ORM, lookup parameters are used to filter querysets based on specific criteria. Here is a list of all the lookup parameters that you can use in Django ORM:

{{

\_\_iexacts : performs a case\_sensitive comparison for an exact match.

\_\_startswith : values that start with a given substring.

\_\_contains : values that contain a given string (case sensitive.)

\_\_icontains : is not case sensitive, it is case insensitive.

}}

exact: Matches exact value.

iexact: Matches exact value, ignoring case.

contains: Matches substring.

icontains: Matches substring, ignoring case.

in: Matches a value in a list of values.

gt: Matches values that are greater than the specified value.

gte: Matches values that are greater than or equal to the specified value.

lt: Matches values that are less than the specified value.

lte: Matches values that are less than or equal to the specified value.

startswith: Matches values that start with a specified value.

istartswith: Matches values that start with a specified value, ignoring case.

endswith: Matches values that end with a specified value.

iendswith: Matches values that end with a specified value, ignoring case.

range: Matches values that fall within a specified range.

date: Matches values on a specific date.

year: Matches values on a specific year.

month: Matches values on a specific month.

day: Matches values on a specific day.

week\_day: Matches values on a specific day of the week.

time: Matches values at a specific time.

isnull: Matches values that are null or not null.

regex: Matches values that match a regular expression.

iregex: Matches values that match a regular expression, ignoring case.

You can use these lookup parameters in conjunction with the field names and comparison operators to filter querysets based on specific criteria.

Different types of datefield in django orm:

In Django's ORM, there are several types of date fields that can be used to represent dates and times in database models. Here are some of the most commonly used date fields in Django:

DateField: This field is used to store a date in the format YYYY-MM-DD.

TimeField: This field is used to store a time in the format HH:MM:SS.

DateTimeField: This field is used to store both a date and a time in the format YYYY-MM-DD HH:MM:SS.

DurationField: This field is used to store a duration or interval of time.

IntervalField: This field is similar to the DurationField, but it is designed to work with PostgreSQL's interval type.

TimeZoneField: This field is used to store a time zone.

YearField: This field is used to store a year in the format YYYY.

YearMonthField: This field is used to store a year and a month in the format YYYY-MM.

MonthField: This field is used to store a month in the format MM.

Each of these fields has its own specific use case and can be used to represent different types of temporal data in a database model.

Values of datefield

auto\_now: If True, the field will be automatically set to the current date every time the object is saved to the database.

auto\_now\_add: If True, the field will be automatically set to the current date when the object is first created and saved to the database.

null: If True, the field is allowed to be null (None).

blank: If True, the field is allowed to be blank (i.e., the empty string "" is a

valid value).

default: The default value for the field if no other value is provided.

Date Methods :

Here are some of the methods that you can use with DateField in Django's ORM:

today(): This method returns the current date, based on the system clock.

year, month, and day: These methods return the year, month, and day components of a date, respectively.

weekday(): This method returns the day of the week as an integer, where Monday is 0 and Sunday is 6.

strftime(): This method formats a date according to a specified format string.

from django.db import modelsclass

Event(models.Model):

name = models.CharField(max\_length=100)

date = models.DateField()

from django.db.models.functions import ExtractWeekDay

# get all events that occurred on a Monday

monday\_events = Event.objects.annotate(weekday=ExtractWeekDay('date')).filter(weekday=0)

# get the year and month of the most recent event

most\_recent\_event = Event.objects.latest('date')

year = most\_recent\_event.date.year

month = most\_recent\_event.date.month

# format the date of an event as a string

event = Event.objects.get(id=1)

date\_string = event.date.strftime('%Y-%m-%d')

joins in orm in django:

Django's ORM supports several types of joins between tables in the database.

You can perform a join between two or more related models using the select\_related() method, which fetches related objects in a single query using a LEFT OUTER JOIN. This method is used to avoid the N+1 query problem that can occur when accessing related objects in a loop.

For example, consider the following models:

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):   
 title = models.CharField(max\_length=100)

author = models.ForeignKey(Author, on\_delete=models.CASCADE)

If you want to fetch all books along with their authors in a single query, you can use the select\_related() method like this:

books = Book.objects.select\_related('author')

for book in books:

print(book.title, book.author.name)

In addition to select\_related(), Django's ORM also supports prefetch\_related(), which performs a separate query to fetch related objects, and annotate(), which allows you to perform aggregate functions across related objects.

features of django orm

Django's ORM (Object-Relational Mapping) provides a number of features that make it easy to work with relational databases using Python. Some of the key features of Django's ORM include:

High-level query API: Django's ORM provides a high-level, Pythonic interface for querying your database that is easy to use and understand. The query API includes methods for filtering, sorting, grouping, and aggregating data, as well as for performing complex queries involving multiple tables and relationships.

Automatic schema generation: Django's ORM can automatically generate database schemas based on your Python models, which makes it easy to create and modify your database schema using Python code.

Database abstraction: Django's ORM provides a database abstraction layer that allows you to write database-agnostic code. This means you can write your code once, and it will work with multiple different database backends without modification.

Object-relational mapping: Django's ORM maps database tables to Python classes and database rows to Python objects, which makes it easy to work with your data using Python code.

Transactions and atomicity: Django's ORM provides support for database transactions and atomicity, which helps ensure that your data remains consistent and reliable even in the face of errors or unexpected events.

Performance optimizations: Django's ORM includes a number of performance optimizations, such as query optimization and caching, that can help improve the performance of your application.

Important:

class Department (models.Model):

    id=models.IntegerField(primary\_key=True)

    name = models.CharField(max\_length=20)

class Person (models.Model):

    name =models.CharField(max\_length=20)

    age =models.IntegerField()

    salary =models.IntegerField()

    dept=models.ForeignKey(Department, on\_delete = models.CASCADE, related\_name='person')

 # get a queryset of all departments and their average salary

dept\_avg\_salary = Department.objects.annotate(avg\_salary=Avg('person\_\_salary'))

# print the average salary for each department

for dept in dept\_avg\_salary:

    print(dept.name, dept.avg\_salary)

Some queries.

# Query all books with a published date greater than or equal to January 1, 2020

books = Book.objects.filter(published\_date\_\_gte='2020-01-01')

# Query all books with a published date greater than January 1, 2020

books = Book.objects.filter(published\_date\_\_gt='2020-01-01')

# Query all books with a published date less than or equal to December 31, 2020

books = Book.objects.filter(published\_date\_\_lte='2020-12-31')

# Query all books with a published date less than December 31, 2020

books = Book.objects.filter(published\_date\_\_lt='2020-12-31')

# Query all books with a title that starts with 'The'

books = Book.objects.filter(title\_\_startswith='The')

# Query all books with a title that starts with 'the' (case-insensitive)

books = Book.objects.filter(title\_\_istartswith='the')

# Query all books with a title that matches a regular expression

books = Book.objects.filter(title\_\_regex=r'^[A-Za-z]+$')

# Query all books with a title that matches a regular expression (case-insensitive)

books = Book.objects.filter(title\_\_iregex=r'^[a-z]+$')

from django.db import models

from django.contrib.auth.models import User

class Post(models.Model):

title = models.CharField(max\_length=200)

content = models.TextField()

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

def \_\_str\_\_(self):

return self.title

class Comment(models.Model):

post = models.ForeignKey(Post, related\_name='comments', on\_delete=models.CASCADE)

user = models.ForeignKey(User, on\_delete=models.CASCADE)

content = models.TextField()

created\_at = models.DateTimeField(auto\_now\_add=True)

updated\_at = models.DateTimeField(auto\_now=True)

def \_\_str\_\_(self):

return f'{self.user.username} on {self.post.title}'

class UserProfile(models.Model):

user = models.OneToOneField(User, on\_delete=models.CASCADE)

bio = models.TextField(blank=True)

birthdate = models.DateField(null=True, blank=True)

profile\_image = models.ImageField(upload\_to='profile\_images/', blank=True)

def \_\_str\_\_(self):

return self.user.username

from django.db.models import Count, Q

posts = Post.objects.all()

# Annotate each post with the number of comments it has

posts = posts.annotate(num\_comments=Count('comments'))

# Prefetch the related comments for each post

posts = posts.prefetch\_related('comments')

# Select the related user for each comment

posts = posts.select\_related('comments\_\_user')

# Filter the posts to only include those with at least 5 comments

posts = posts.filter(num\_comments\_\_gte=5)

# Filter the comments to only include those made by a specific user

posts = posts.filter(comments\_\_user\_\_username='someusername')

# Order the posts by the number of comments they have, in descending order

posts = posts.order\_by('-num\_comments')