В этой статье я отмечу несколько основных способ оптимизировать orm запросы. Сделаю 3 таблицы для этого:

!@#$

from django.contrib.auth import get\_user\_model  
from django.db import models  
  
UserModel = get\_user\_model()  
  
  
class ArticleAuthorModel(models.Model):  
 author = models.ForeignKey(UserModel, models.CASCADE)  
 article = models.ForeignKey("ArticleModel", models.CASCADE)  
  
  
class ArticleModel(models.Model):  
 title = models.CharField(max\_length=80)  
 email = models.CharField(max\_length=50)  
 author = models.ManyToManyField(UserModel, "articles", through=ArticleAuthorModel)  
  
 class Meta:  
 db\_table = "ArticleModel"  
  
  
class CommentsModel(models.Model):  
 text = models.TextField()  
 article = models.ForeignKey(ArticleModel, models.CASCADE, "comments")  
 author = models.ForeignKey(UserModel, models.CASCADE, "comments")  
  
 class Meta:  
 db\_table = "CommentsModel"

!@#$

Не стоит забывать, что методы вроде filter()и exclude()на самом деле не запускают запрос — они настраивают queryset для запуска при его итерации. Запрос выполняется лишь при прямом обращении к данным. Т.е. в строке:

!@#$

result = CommentsModel.objects.all()

!@#$

Не выполняется запрос. Переменная result только хранит запрос, а не сами данные

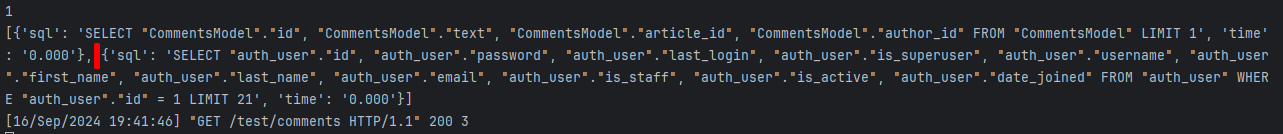
Итак, первая заметка – использование select\_related. Этот метод решает проблему N+1, добавляя inner join.

Без select\_related:

!@#$  
class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = CommentsModel.objects.all()  
 print(result[0].author.id)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

Результат



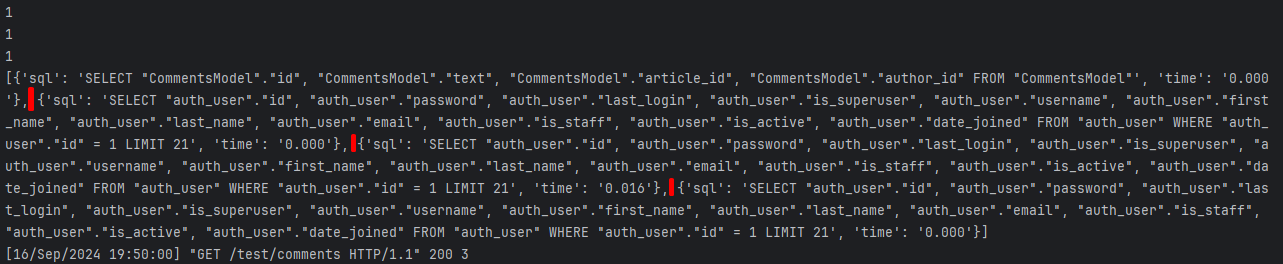
Вывод: 2 запроса. Один для получения данных из указанной модели и один для получения данных из связанной модели. Можно изменить код так, чтобы получить все комментарии:

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = CommentsModel.objects.all()  
 for res in result:  
 print(res.author.id)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

Результат



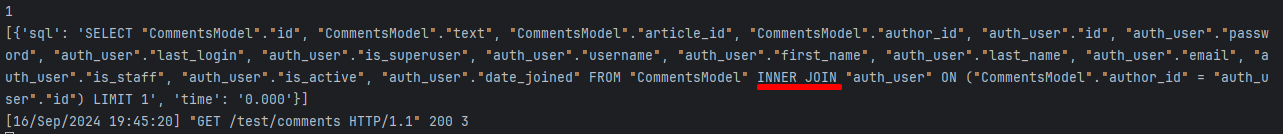
Вывод: один запрос для получения всех комментариев, который вернул 3 записи, и еще 3 запроса для получения автора каждого комментария. Таким темпом, если надо будет получить 1000 записей, то выполнится еще 1000 запросов. Это и есть проблема N+1. Добавляю в запрос select\_related:

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = CommentsModel.objects.select\_related("author").all()  
 print(result[0].author.id)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

Результат:



Теперь для все данные получаются в одном запросе через inner join

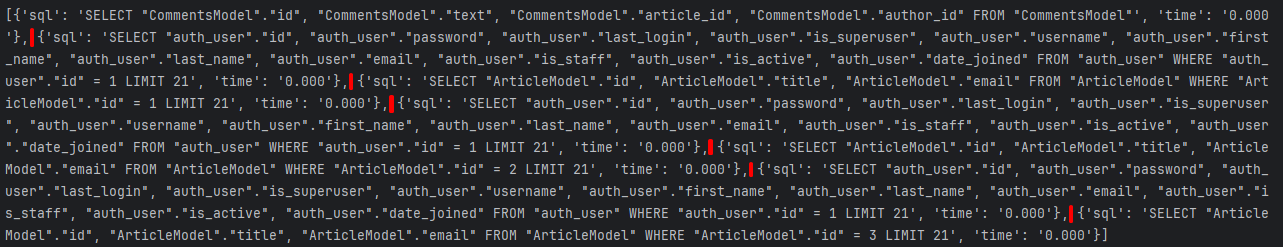
Если добавить в изначальный запрос еще инфу о статье:

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = CommentsModel.objects.all()  
 for res in result:  
 print(res.author.id)  
 print(res.article.id)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

то на выхлопе получим 7 запросов

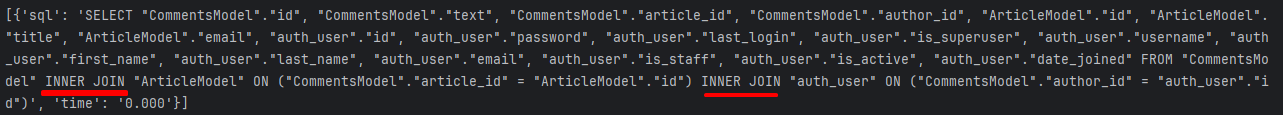


Для такого случая в related\_related можно указывать несколько таблиц:

!@#$  
class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = CommentsModel.objects.select\_related("author", "article").select\_related().all()  
 for res in result:  
 print(res.author.id)  
 print(res.article.id)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

и получаем 1 запрос с двумя inner join:



related\_related можно использовать при запросах ForeignKey, OneToOne.

При запросах ManyToMany или обратных ForeignKey следует использовать prefetch\_related.

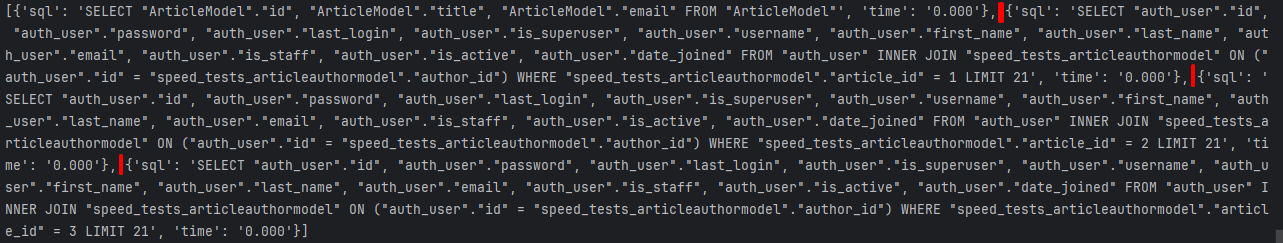
Без него:

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = ArticleModel.objects.all()  
 for res in result:  
 print(res.author.all())  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

Результат:

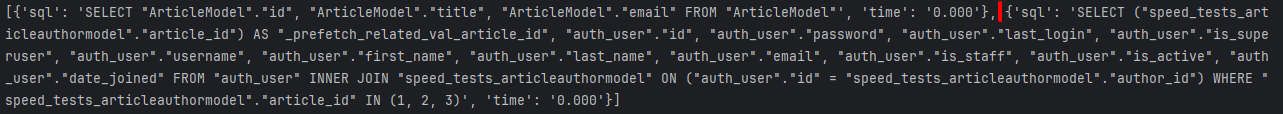


С использованием prefetch\_related:

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = ArticleModel.objects.all()  
 for res in result:  
 print(res.author.all())  
 print(connection.queries)  
 return HttpResponse(200)

!@#$



Запросов стало 2, а не 1. Дело в том, что prefetch\_related не использует под капотом inner join. Он делает столько запросов в БД, сколько таблиц задействовано – в каждую отдельно и на уровне самого питона уже объединит данные­

Для большего контроля можно использовать объект Prefetch:

!@#$

class MyView(View):

def get(self, request, \*args, \*\*kwargs):

result = ArticleModel.objects.prefetch\_related(

Prefetch("author", queryset=UserModel.objects.order\_by("-date\_joined"))

).all()

for res in result:

print(res.author.all())

print(connection.queries)

return HttpResponse(200)

!@#$

В этом запросе записи из связанной модели отсортированы по дате. По сути сюда можно подсунуть любой запрос: annotate, aggregate или еще один prefetch\_related даже, чтобы связать таблицы на большем уровни вложенности.

Не в каждом запросе будут требовать все поля данных. Поэтому, можно использовать метод only():

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = UserModel.objects.only("id", "username")  
 for res in result:  
 print(res)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

Результат:



Если запрос возвращает много записей, то можно брать их частями, использую принцип пагинации:

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 result = CommentsModel.objects.all()[0:1]  
 for res in result:  
 print(res)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

Результат:



Вместо трёх объектов запрос вернул 1, как и было указано. В запросе можно указать параметры запроса, передавая какая страница пагинации была запрошена:

!@#$

<http://127.0.0.1:8080/test/comments?offset=2&limit=3>

!@#$

вью:

!@#$

class MyView(View):  
  
 def get(self, request, \*args, \*\*kwargs):  
 offset = int((request.GET.get("offset")))  
 limit = int((request.GET.get("limit")))  
  
 result = CommentsModel.objects.all()[offset:offset+limit]  
 for res in result:  
 print(res)  
 print(connection.queries)  
 return HttpResponse(200)

!@#$

Это самые базовые действия для оптимизации запросов. Помимо них можно назвать кеширование, индексацию и