<https://djbook.ru/rel3.0/#the-model-layer>

QuerySets – ленивы, создание [QuerySet](https://djbook.ru/rel3.0/ref/models/querysets.html#django.db.models.query.QuerySet) не выполняет запросов к базе данных. Вы можете добавлять фильтры хоть весь день и Django не выполнит ни один запрос, пока [QuerySet](https://djbook.ru/rel3.0/ref/models/querysets.html#django.db.models.query.QuerySet) не вычислен

get()

filter()

all()

exclude()

model\_name()

save()

delete()

[::]

Further filtering or ordering of a sliced queryset is prohibited due to the ambiguous nature of how that might work.

Отрицательные индексы (например, Entry.objects.all()[-1]) не поддерживаются.

Django предоставляет [класс F](https://djbook.ru/rel3.0/ref/models/expressions.html#django.db.models.F) для таких сравнений. Экземпляр F() рассматривается как ссылка на другое поле модели. Эти ссылки могут быть использованы для сравнения значений двух разных полей одного объекта модели.

>>> **from** django.db.models **import** F

>>> Entry**.**objects**.**filter(number\_of\_comments\_\_gt**=**F('number\_of\_pingbacks'))

Вы можете использовать два нижних подчеркивания для использования полей связанных объектов в F(). Объект F() с двойным нижним подчеркиванием обеспечит все необходимые JOIN для получения необходимых связанных объектов. Например, чтобы получить все записи, у которых имя автора совпадает с названием блога, нужно выполнить такой запрос:

>>> Entry**.**objects**.**filter(authors\_\_name**=**F('blog\_\_name'))

**Caching and**QuerySet**s**

Manager.raw()

Если вам необходимо выполнить более сложные вопросы (например, запросы с OR), вы можете использовать [объект Q](https://djbook.ru/rel3.0/ref/models/querysets.html#django.db.models.Q).

<https://djbook.ru/rel3.0/topics/db/aggregation.html>

**class** **Book**(models**.**Model):

name **=** models**.**CharField(max\_length**=**300)

pages **=** models**.**IntegerField()

price **=** models**.**DecimalField(max\_digits**=**10, decimal\_places**=**2)

rating **=** models**.**FloatField()

authors **=** models**.**ManyToManyField(Author)

publisher **=** models**.**ForeignKey(Publisher, on\_delete**=**models**.**CASCADE)

pubdate **=** models**.**DateField()

Book**.**objects**.**aggregate(Avg('price'), Max('price'), Min('price'))

{'price\_\_avg': 34.35, 'price\_\_max': Decimal('81.20'), 'price\_\_min': Decimal('12.99')}

>>> Book**.**objects**.**aggregate(average\_price**=**Avg('price'))

{'average\_price': 34.35}

>>> q **=** Book**.**objects**.**annotate(num\_authors**=**Count('authors'))

>>> q[0]**.**num\_authors

2

>>> q[1]**.**num\_authors

1

>>> Store**.**objects**.**annotate(min\_price**=**Min('books\_\_price'), max\_price**=**Max('books\_\_price'))

>>> Store**.**objects**.**aggregate(youngest\_age**=**Min('books\_\_authors\_\_age'))

**Использование обратных связей**

>>> Publisher**.**objects**.**aggregate(oldest\_pubdate**=**Min('book\_\_pubdate'))

>>> Author**.**objects**.**annotate(total\_pages**=**Sum('book\_\_pages'))

**Аннотация агрегации**

>>> Book**.**objects**.**annotate(num\_authors**=**Count('authors'))**.**aggregate(Avg('num\_authors'))

{'num\_authors\_\_avg': 1.66}

<https://djbook.ru/rel3.0/ref/models/querysets.html>

Этот пример отображает разницу между обычной выборкой и с select\_related(). Обычная выборка:

*# Hits the database.*

e **=** Entry**.**objects**.**get(id**=**5)

*# Hits the database again to get the related Blog object.*

b **=** e**.**blog

И с select\_related:

*# Hits the database.*

e **=** Entry**.**objects**.**select\_related('blog')**.**get(id**=**5)

*# Doesn't hit the database, because e.blog has been prepopulated*

*# in the previous query.*

b **=** e**.**blog

*# Find all the blogs with entries scheduled to be published in the future.*

blogs **=** set()

**for** e **in** Entry**.**objects**.**filter(pub\_date\_\_gt**=**timezone**.**now())**.**select\_related('blog'):

*# Without select\_related(), this would make a database query for each*

*# loop iteration in order to fetch the related blog for each entry.*

blogs**.**add(e**.**blog)

Порядок использования filter() и select\_related() не важен. Эти экземпляры QuerySet одинаковы:

*# Hits the database with joins to the author and hometown tables.*

b **=** Book**.**objects**.**select\_related('author\_\_hometown')**.**get(id**=**4)

p **=** b**.**author *# Doesn't hit the database.*

c **=** p**.**hometown *# Doesn't hit the database.*

select\_related('foo', 'bar') аналогично select\_related('foo').select\_related('bar').

select\_related работает только со связями возвращающими один объект - внешний ключ и связь один-к-одному.

Pizza.objects.all() выполнит запрос к таблице Topping для **каждого** объекта Pizza в QuerySet.

>>> Pizza**.**objects**.**all()

["Hawaiian (ham, pineapple)", "Seafood (prawns, smoked salmon)"...

Мы можем уменьшить количество запросов до двух, используя prefetch\_related:

>>> Pizza**.**objects**.**all()**.**prefetch\_related('toppings')

Помните, для всех QuerySets любой последующий метод, который изменяет запрос, выполнит новый запрос в бд.

>>> pizzas **=** Pizza**.**objects**.**prefetch\_related('toppings')

>>> [list(pizza**.**toppings**.**filter(spicy**=True**)) **for** pizza **in** pizzas]

>>> Restaurant**.**objects**.**prefetch\_related('pizzas\_\_toppings')

Этот запрос выполнит предварительную загрузку всех пицц(Pizza) для ресторанов(Restaurant) и всех ингредиентов(Topping) для пицц. В результате будет выполнено 3 запроса - один для Restaurant, один для Pizza, и один для Topping.

или вот так за 2 запроса:  
>>> Restaurant**.**objects**.**select\_related('best\_pizza')**.**prefetch\_related('best\_pizza\_\_toppings')