<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')

<https://concisecoder.io/2018/11/04/django-orm-optimization-tips/>

**Profile**

**Be aware of QuerySet’s lazy evaluation.**

**Be aware of which model attributes are not cached.**

1. # Foreign-key related objects
2. person = Person.objects.get(id=1)
3. person.father # foreign object is retrieved and cached
4. person.father # cached version is used
5. ## Never cached
6. # Callable attributes
7. person = Person.objects.get(id=1)
8. person.children.all() # Database hit
9. person.children.all() # Another database hit

**Use select\_related() and prefetch\_related() when you will need foreign-key/reverse related objects.**

**Try to avoid database queries in a loop.**

**Use iterator() to iterate through a very large QuerySet only once.**

**Do work in the database rather than in Python.**

**Use values() and values\_list() to get only the things you need.**

>>> Blog**.**objects**.**values()

<QuerySet [{'id': 1, 'name': 'Beatles Blog', 'tagline': 'All the latest Beatles news.'}]>

>>> Blog**.**objects**.**values('id', 'name')

<QuerySet [{'id': 1, 'name': 'Beatles Blog'}]>

>>> Entry**.**objects**.**values\_list('id', 'headline')

<QuerySet [(1, 'First entry'), ...]>

>>> **from** django.db.models.functions **import** Lower

>>> Entry**.**objects**.**values\_list('id', Lower('headline'))

<QuerySet [(1, 'first entry'), ...]>

>>> Entry**.**objects**.**values\_list('id')**.**order\_by('id')

<QuerySet[(1,), (2,), (3,), ...]>

>>> Entry**.**objects**.**values\_list('id', flat**=True**)**.**order\_by('id')

<QuerySet [1, 2, 3, ...]>

>>> Entry**.**objects**.**values\_list('id', 'headline', named**=True**)

<QuerySet [Row(id=1, headline='First entry'), ...]>

**Use defer() and only() when you only need certain fields.**

Use defer():

1. queryset = Person.objects.defer('age') # Imagine age is computationally expensive
2. **for** person **in** queryset:
3. **print**(person.id)
4. **print**(person.name)

Use only():

1. queryset = Person.objects.only('name')
2. **for** person **in** queryset:
3. **print**(person.name)

**Use count() and exists() when you don’t need the contents of the QuerySet.**

Use count():

1. # DON'T
2. count = len(Person.objects.all()) # Evaluates the entire queryset
3. # DO
4. count = Person.objects.count() # Executes more efficient SQL to determine count

Use exists():

1. # DON'T
2. exists = len(Person.objects.all()) > 0
3. # DO
4. exists = Person.objects.exists()

**Use delete() and update() when possible.**

**Use bulk\_create() when possible.**

1. names = ['Jeff', 'Beth', 'Tim']
2. creates = []
3. **for** name **in** names:
4. creates.append(
5. Person(name=name, age=0)
6. )
7. Person.objects.bulk\_create(creates)

Similarly, bulk-add to many-to-many fields:

1. person = Person.objects.get(id=1)
2. person.jobs.add(job1, job2, job3)

**Use foreign key values directly.**

1. # DON'T
2. father\_id = Person.objects.get(id=1).father.id # Causes a needless database query
3. # DO
4. father\_id = Person.objects.get(id=1).father\_id # The foreign key is already cached. No query

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

* [migrate](https://djbook.ru/rel3.0/ref/django-admin.html#django-admin-migrate), which is responsible for applying and unapplying migrations.
* [makemigrations](https://djbook.ru/rel3.0/ref/django-admin.html#django-admin-makemigrations), которая отвечает за создание новых миграций на основе изменений в моделях.
* [sqlmigrate](https://djbook.ru/rel3.0/ref/django-admin.html#django-admin-sqlmigrate), которая выводит SQL запросы для миграции.
* [showmigrations](https://djbook.ru/rel3.0/ref/django-admin.html#django-admin-showmigrations), which lists a project’s migrations and their status.

Reverting migrations

Any migration can be reverted with [migrate](https://djbook.ru/rel3.0/ref/django-admin.html#django-admin-migrate) by using the number of previous migrations:

$ python manage.py migrate books 0002

Operations to perform:

Target specific migration: 0002\_auto, from books

Running migrations:

Rendering model states... DONE

Unapplying books.0003\_auto... OK

If you want to revert all migrations applied for an app, use the name zero:

$ python manage.py migrate books zero

Operations to perform:

Unapply all migrations: books

Running migrations:

Rendering model states... DONE

Unapplying books.0002\_auto... OK

Unapplying books.0001\_initial... OK

squash:

./manage.py squashmigrations myapp 0004

Will squash the following migrations:

- 0001\_initial

- 0002\_some\_change

- 0003\_another\_change

- 0004\_undo\_something

<https://russianblogs.com/article/2217508687/>

**python manage.py makemigrations –merge**

Объединение будет работать только в том случае, если операции, не конфликтуют друг с другом (работают на разных полях или моделях)