Работа с запросами

В Django работа с базой данных осуществляется через **ORM (Object-Relational Mapper)**. Вместо написания SQL-запросов вручную вы оперируете Python-объектами и специальными коллекциями, называемыми QuerySet.

Главные особенности QuerySet

- ✓ Ленивость (Lazy evaluation). Сам по себе запрос к базе не отправляется в момент создания переменной. Вы можете фильтровать и сортировать данные сколько угодно, но Django пойдет в БД только тогда, когда вам реально понадобятся данные (например, при цикле for, преобразовании в список или выводе в шаблон).
- ✓ Цепочки методов (Method Chaining). Большинство методов QuerySet возвращают новый QuerySet. Это позволяет выстраивать длинные цепочки: `User.objects.filter(active=True).order_by('name').`
- ✓ Кэширование. Как только QuerySet «вычисляется», он сохраняет результат в себе. Повторное использование той же переменной не вызовет новый запрос к базе.

Модели для работы с запросами

```
from datetime import date
from django.db import models

  class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()
    def __str__(self):
        return self.name

  class Author(models.Model):
    name = models.CharField(max_length=200)
    email = models.EmailField()
    def __str__(self):
        return self.name

  class Entry(models.Model):
    blog = models.ForeignKey(Blog, on_delete=models.CASCADE)
    headline = models.CharField(max_length=255)
    body_text = models.TextField()
    pub_date = models.DateField()
    mod_date = models.DateField(default=date.today)
    authors = models.ManyToManyField(Author)
    number_of_comments = models.IntegerField(default=0)
    number_of_pingbacks = models.IntegerField(default=0)
    rating = models.IntegerField(default=5)
    def __str__(self):
          return self.headline
```

CRUD-операции

В контексте Django QuerySet, CRUD (Create, Read, Update, Delete) — это четыре базовые операции для работы с данными.

Create (Создание)

- Быстро, сразу сохраняет в БД

```
blog = Blog.objects.create(name='TechHub', tagline='Код и железо')
```

- Нужна дополнительная логика перед сохранением (переопределение save())

```
author = Author(name='Анна', email='anna@mail.ru')
author.save()
```

- bulk_create — это мощный инструмент для **массового создания** записей в базе данных одним SQL-запросом. Это в десятки и сотни раз быстрее, чем вызывать .save() или .create() в цикле, так как количество обращений к БД сводится к минимуму. Но! **Метод save() не вызывается**: Логика, написанная внутри переопределенного метода .save(), будет проигнорирована.

```
Entry.objects.bulk_create([Entry(...), )])
```

CRUD-операции

ManyToMany: связи добавляются **только после** сохранения основного объекта

```
entry = Entry.objects.create(
    blog=blog, headline='Django 6.0', body_text='...', pub_date='2026-01-01'
)
# .add(), .set() или .remove() работают только после .save() или .create()
entry.authors.add(author1, author2)
# или
entry.authors.set([author1.id, author2.id])  # перезаписывает связи
```

CRUD-операции

В контексте Django QuerySet, CRUD (Create, Read, Update, Delete) — это четыре базовые операции для работы с данными.

Read (Чтение)

Метод	Возвращает	Поведение
.get()	Один объект	Формирует исключение DoesNotExist (0 шт) или MultipleObjectsReturned (>1 шт)
.filter()	QuerySet	Всегда возвращает коллекцию (даже если 0 или 1 объект)
.first() / .last()	Один объект или None	Безопасно, не формирует исключений
.exists()	bool	Работает мгновенно

```
# Безопасное чтение
entry = Entry.objects.filter(id=1).first()  # вернёт None, если нет

# Фильтрация по связям
entries_by_author = Entry.objects.filter(authors__email__endswith='@qq.com')

# Проверка наличия без загрузки данных
has_high_rated = Entry.objects.filter(rating=5).exists()
```

CRUD-операции

В контексте Django QuerySet, CRUD (Create, Read, Update, Delete) — это четыре базовые операции для работы с данными.

Update (Обновление)

Способ	Пример
Экземпляр + .save()	entry.rating = 4 entry.save()
QuerySet .update()	Entry.objects.filter(id=1).update(rating=4)
bulk_update()	Entry.objects.bulk_update(entries, ['rating', 'mod_date'])

CRUD-операции

В контексте Django QuerySet, CRUD (Create, Read, Update, Delete) — это четыре базовые операции для работы с данными.

Delete (Удаление)

Экземпляр .delete()	entry.delete()	Удаляет объект
QuerySet .delete()	Entry.objects.filter(rating=1).delete()	Массовое удаление, **НЕ** вызывает .delete() на каждом объекте
Каскадное удаление	blog.delete()	Так как blog = models.ForeignKey(..., on_delete=models.CASCADE), удалятся все связанные Entry

Фильтрация и условия

Когда вы вызываете .filter(), запрос к БД не выполняется. Django работает по принципу Lazy evaluation :

- Сборка дерева условий. Каждый вызов .filter() добавляет узел во внутренний объект Query.
- Разрешение связей. При наличии __ (например, blog__name) Django находит FK/M2M, определяет целевую таблицу и планирует JOIN.
- Компиляция в SQL. При вычислении QuerySet (for, list(), len(), first()) QueryCompiler:
  - Строит FROM + JOIN
  - Формирует WHERE из дерева условий
  - Добавляет GROUP BY/HAVING (если есть агрегации)
  - Отправляет запрос в БД

Ключевое правило: Все условия, переданные в один .filter(), соединяются через AND. Для OR/NOT используются Q-объекты.

Поиск по полям: `field__lookuptype=value`

Фильтрация и условия. Field lookups

Lookup	SQL-аналог	Пример	Примечание
exact	=	Entry.objects.filter(headline__exact='Django 5')	**По умолчанию** field='val' это field__exact='val'
iexact	ILIKE / = (case-insensitive)	Blog.objects.filter(name__iexact='techhub')	Без учёта регистра
gt	>	Entry.objects.filter(rating__gt=4)	
gte	>=	Entry.objects.filter(number_of_comments__gte=100)	
lt	<	Entry.objects.filter(pub_date__lt='2025-01-01')	
lte	<=	Entry.objects.filter(rating__lte=3)	
in	IN (...)	Entry.objects.filter(rating__in=[1, 5])	Принимает список, кортеж или QuerySet
range	BETWEEN ... AND ...	Entry.objects.filter(pub_date__range=['2025-01-01', '2025-12-31'])	Включительно с обеих сторон

Фильтрация и условия. Field lookups

Lookup	SQL-аналог	Пример	Примечание
contains	LIKE '%...%'	Entry.objects.filter(body_text__contains='Python')	Регистрозависимо
icontains	ILIKE '%...%'	Entry.objects.filter(headline__icontains='django')	Без учёта регистра
startswith	LIKE '...%'	Author.objects.filter(email__startswith='admin@')	**Использует индекс**
istartswith	ILIKE '...%'	Blog.objects.filter(name__istartswith='dev')	
endswith	LIKE '%...'	Author.objects.filter(email__endswith='@gmail.com')	Не использует индекс
iendswith	ILIKE '%...'	Entry.objects.filter(headline__iendswith='news')	
regex	REGEXP / ~	Entry.objects.filter(headline__regex=r'^[A-Z]')	Зависит от диалекта БД
iregex	IREGEXP / ~*	Entry.objects.filter(body_text__iregex=r'\bpython\b')	

Фильтрация и условия. Field lookups

Lookup	Пример	Что делает
year	Entry.objects.filter(pub_date__year=2025)	Год
month	Entry.objects.filter(pub_date__month=6)	Месяц (1-12)
day	Entry.objects.filter(pub_date__day=15)	День месяца
week	Entry.objects.filter(pub_date__week=25)	Номер недели года
week_day	Entry.objects.filter(pub_date__week_day=2)	День недели (1=Вс, 2=Пн ... 7=Сб)
quarter	Entry.objects.filter(pub_date__quarter=3)	Квартал (1-4)
hour / minute / second	Entry.objects.filter(mod_date__hour=14)	Время из DateTimeField
date	Entry.objects.filter(mod_date__date=date.today())	Извлекает дату из DateTimeField для сравнения

Фильтрация и условия. Field lookups

Lookup	Пример	Поведение
isnull	Entry.objects.filter(blog__isnull=False)	True → IS NULL, False → IS NOT NULL
isnull на FK/M2M	Blog.objects.filter(entry__isnull=True)	Находит блоги **без записей**
isempty	Entry.objects.filter(authors__isempty=True)	Только для PostgreSQL ArrayField / JSONField

Фильтрация и условия. Связи

Django позволяет "ходить" по связям в любую сторону через ___. По умолчанию .filter() по связям использует INNER JOIN

Тип связи	Направление	Пример запроса	Что делает Django
ForeignKey (прямая)	Entry → Blog	Entry.objects.filter(blog__tagline__icontains='python')	INNER JOIN app_blog
ForeignKey (обратная)	Blog → Entry	Blog.objects.filter(entry__rating=5)	INNER JOIN app_entry
ManyToMany (прямая)	Entry → Author	Entry.objects.filter(authors__email__endswith='@mail.ru')	JOIN app_entry_authors + JOIN app_author
ManyToMany (обратная)	Author → Entry	Author.objects.filter(entry__headline__startswith='Review')	Те же JOIN, но в обратную сторону
Цепочка (любая глубина)	Blog → Entry → Author	Blog.objects.filter(entry__authors__name='Алиса', entry__rating__gte=4)	Два JOIN, одно условие на entry, другое на author

Сортировка, слайсинг и уникальность

Сортировка (order_by)

```
# По возрастанию
Entry.objects.order_by('pub_date')
# По убыванию
Entry.objects.order_by('-rating')
# Несколько полей (второе сортируется при равенстве первого)
Entry.objects.order_by('blog__name', '-pub_date’)
```

Слайсинг ([:n], [n:m]). Django транслирует слайсинг в LIMIT и OFFSET

```
Entry.objects.order_by('-pub_date')[:10]      # LIMIT 10
Entry.objects.order_by('-pub_date')[10:20]    # LIMIT 10 OFFSET 10
```

Уникальность (distinct()). Убирает дубликаты строк. Особенно важно после JOIN (фильтрация по M2M или обратным FK):

```
# Без distinct(): блог вернётся N раз (по количеству записей с rating=5)
Blog.objects.filter(entry__rating=5)
# С distinct(): каждый блог только один раз
Blog.objects.filter(entry__rating=5).distinct()
```

Работа со связями

Работа со связями в Django ORM строится вокруг трёх концепций: **прямой доступ**, **обратный доступ** и **оптимизация запросов**.

Поле	Тип	Направление	SQL-механика
Entry.blog	ForeignKey	Entry → Blog (ManyToOne)	Внешний ключ blog_id в таблице entry
Entry.authors	ManyToMany	Entry ↔ Author	Промежуточная таблица entry_authors (создаётся автоматически)

Прямой доступ

```
entry = Entry.objects.get(id=1)
# ForeignKey → возвращает объект (или вызывает SELECT при первом обращении)
entry.blog.name
entry.blog.tagline
# ManyToMany → возвращает RelatedManager (подзапрос)
entry.authors.all()          # QuerySet авторов
entry.authors.filter(email__endswith='@gmail.com')
```

Работа со связями

Обратный доступ

```
blog = Blog.objects.get(id=1)
blog.entry_set.all()          # Все записи этого блога
author = Author.objects.get(id=1)
author.entry_set.all()        # Все записи этого автора
```

Всегда задавайте related_name в полях, чтобы писать читаемый код!

```
# В модели Entry:
blog = models.ForeignKey(Blog, on_delete=models.CASCADE, related_name='entries')
authors = models.ManyToManyField(Author, related_name='entries')
# Теперь:
blog.entries.all()
author.entries.filter(rating=5)
```

Оптимизация запросов (борьба с N+1)

Проблема N+1 — это «тихий убийца» производительности Django-приложений. Она возникает, когда вы в цикле обращаетесь к связанным данным (ForeignKey или ManyToMany), которые не были загружены заранее.

```
for e in Entry.objects.all():
    print(e.blog.name)
```

1 запрос для получения публикаций + 100 запросов для получения названия блога **= 101 запрос**.

Если на странице 1000 публикаций в блогах — это 1001 запрос. База данных «ложится» от огромного количества мелких однотипных обращений.

Зачем это оптимизировать?

Скорость загрузки страницы. Время выполнения одного SQL-запроса может быть маленьким, но суммарная задержка 100+ запросов делает сайт «тормознутым».

Экономия ресурсов БД. Базе данных проще выполнить один сложный JOIN, чем сотни простых. Это освобождает ресурсы для других пользователей.

Масштабируемость. При 10 пользователях сайт может работать нормально, но при 1000 — N+1 гарантированно приведет к отказу сервера.

Оптимизация запросов (борьба с N+1)

Проблема N+1 — это «тихий убийца» производительности Django-приложений. Она возникает, когда вы в цикле обращаетесь к связанным данным (ForeignKey или ManyToMany), которые не были загружены заранее.

```
for e in Entry.objects.all():
    print(e.blog.name)
```

1 запрос для получения публикаций + 100 запросов для получения названия блога = **101 запрос**.

Если на странице 1000 публикаций в блогах — это 1001 запрос. База данных «ложится» от огромного количества мелких однотипных обращений.

Зачем это оптимизировать?

Скорость загрузки страницы. Время выполнения одного SQL-запроса может быть маленьким, но суммарная задержка 100+ запросов делает сайт «тормознутым».

Экономия ресурсов БД. Базе данных проще выполнить один сложный JOIN, чем сотни простых. Это освобождает ресурсы для других пользователей.

Масштабируемость. При 10 пользователях сайт может работать нормально, но при 1000 — N+1 гарантированно приведет к отказу сервера.

Решение:

select_related() → для ForeignKey / OneToOne

prefetch_related() → для ManyToMany / обратных ForeignKey

Оптимизация запросов (борьба с N+1)

select_related() → для ForeignKey / OneToOne

Делает **один запрос с JOIN**

```
# Плохо: 1 запрос на получение записей + N запросов к Blog
for e in Entry.objects.all():
    print(e.blog.name)
# Хорошо: 1 запрос с INNER JOIN
entries = Entry.objects.select_related('blog').all()
for e in entries:
    print(e.blog.name)  # Данные уже в памяти
```

Оптимизация запросов (борьба с N+1)

prefetch_related() для ManyToMany / обратных ForeignKey

Делает **2 запроса**: основной + SELECT ... WHERE id IN (...), соединяет в Python

```
#  Плохо: N запросов к таблице authors
for e in Entry.objects.all():
    print(e.authors.all())


#  Хорошо: 2 запроса независимо от N
entries = Entry.objects.prefetch_related('authors').all()
for e in entries:
    print(e.authors.all())
```

Комбинирование

```
# Загружаем блог (JOIN) + авторов (2 запроса) + комментарии к записям (ещё 2
  запроса)
qs = Entry.objects.select_related('blog').prefetch_related('authors', 'comments')
```

Q-объекты (логические условия)

Q-объекты (django.db.models.Q) — это инструмент для построения сложных запросов с логическими операторами ИЛИ, И, НЕ. Обычный .filter(a=1, b=2) всегда соединяет условия через AND. Q ломает это ограничение и позволяет собирать динамические, вложенные и комбинируемые условия.

Q-объекты поддерживают побитовые операторы (именно их, а не and/or)

Оператор	Логика	Пример
|	OR	Q(rating=5) | Q(number_of_comments__gt=100)
&	AND	Q(rating__gte=4) & Q(pub_date__year=2025)
~	NOT	~Q(blog__name='Спам')

Q-объекты (логические условия)

Простое OR

```
from django.db.models import Q
# Записи с рейтингом 5 ИЛИ >100 комментариев
Entry.objects.filter(Q(rating=5) | Q(number_of_comments__gt=100))
```

AND + NOT + обычные kwargs: обычные аргументы в .filter() соединяются через AND со всем Q-деревом.

```
# Рейтинг 4 или 5, НО НЕ из блога "Тестовый", И ТОЛЬКО за 2025 год
Entry.objects.filter(
    Q(rating=4) | Q(rating=5), ~Q(blog__name='Тестовый'),pub_date__year=2025 )
```

По связям (ForeignKey / ManyToMany)

```
# Авторы, писавшие записи про "Django" ИЛИ "Python"
Author.objects.filter(
    Q(entry__headline__icontains='Django') |
    Q(entry__headline__icontains='Python')
).distinct()  # distinct() обязателен при OR по M2M/обратным FK
# Блоги, у которых НЕТ записей с рейтингом 1
Blog.objects.filter(~Q(entry__rating=1)).distinct()
```

F-выражения

F-выражения (django.db.models.F) — это механизм ссылки на значения полей модели **на стороне базы данных**, а не в Python. Они позволяют сравнивать поля между собой, выполнять арифметику, обновлять значения атомарно и строить динамические вычисления прямо в SQL.

Проблема без F()	Решение с F()
entry.comments = entry.comments + 1 → **Race Condition** при конкурентных запросах	update(comments=F('comments')+1) → атомарный UPDATE ... SET comments = comments + 1
Сравнение полей в Python: if entry.comments > entry.pingbacks: → загружает всё в память	filter(comments__gt=F('pingbacks')) → выполняется на стороне БД через WHERE comments > pingbacks

F-выражения

Сравнение полей одной модели

```
from django.db.models import F
# Записи, где комментариев больше, чем пингбэков
Entry.objects.filter(number_of_comments__gt=F('number_of_pingbacks'))

# Записи с рейтингом выше, чем средний по блогу (через подзапрос)
from django.db.models import Avg
Entry.objects.filter(rating__gt=Avg('blog__entry__rating’))
```

Арифметика при обновлении (UPDATE)

```
# Атомарный инкремент (без гонок потоков)
Entry.objects.filter(id=1).update(number_of_comments=F('number_of_comments') + 1)

# Обновление нескольких полей
Entry.objects.filter(rating__lt=3).update(
    rating=F('rating') + 1,
    mod_date=F('pub_date') + timedelta(days=7)  # Django автоматически транслирует
  timedelta
)
```

F-выражения

Атомарность и Race Conditions (Ключевое преимущество)

```
# ОПАСНО: два процесса читают 5, оба пишут 6. Одно обновление теряется.
entry = Entry.objects.get(id=1)
entry.number_of_comments += 1
entry.save()

 БЕЗОПАСНО: БД выполнит SET comments = comments + 1 атомарно
Entry.objects.filter(id=1).update(number_of_comments=F('number_of_comments') + 1)
```

Для счётчиков **всегда** используйте F() в update().

Агрегация и аннотация

В Django ORM агрегация и аннотация используются для выполнения вычислений на стороне базы данных. Главное различие между ними заключается в том, **на каком уровне** производятся расчеты: для всего набора данных сразу или для каждого объекта в отдельности.

Агрегация (aggregate())

Этот метод вычисляет значения для **всего QuerySet** и возвращает словарь (dict) с результатами. После вызова aggregate() вы получаете итоговое число, а не список объектов, поэтому этот метод является терминальным (завершающим запрос).

Аннотация (annotate())

Этот метод добавляет вычисляемое поле к **каждому объекту** в QuerySet. Результатом остается QuerySet, который можно фильтровать или сортировать дальше.

Результат	Словарь (dict)	Набор объектов (QuerySet)
Область действия	Весь QuerySet целиком	Каждый объект по отдельности
Цепочка вызовов	Завершает запрос	Можно продолжать (filter, order_by)

Агрегация и аннотация

Для обоих методов используются функции из django.db.models:

- ✓ Count — подсчет количества.
- ✓ Sum — суммирование значений.
- ✓ Avg — среднее арифметическое.
- ✓ Min / Max — поиск минимального или максимального значения.

Важный нюанс: порядок вызова filter() и annotate() имеет значение. Если вы фильтруете после аннотации, фильтр применится к вычисленному значению. Если до — аннотация будет считаться только по отфильтрованным записям.

Агрегация и аннотация

```python
from django.db.models import Count, Avg, Sum, Max


# Одна цифра: сколько всего записей?
total = Entry.objects.aggregate(total=Count('id'))['total']


# Средний рейтинг всех записей
avg_rating = Entry.objects.aggregate(avg=Avg('rating'))['avg']


# Аннотация: сколько записей у каждого блога?
blogs_with_count = Blog.objects.annotate(entry_count=Count('entry'))
# Теперь у каждого blog есть атрибут .entry_count


# Аннотация: средний рейтинг авторов + фильтрация по нему
from django.db.models import Q
top_authors = Author.objects.annotate(
    avg_entry_rating=Avg('entry__rating'),
    total_entries=Count('entry')
).filter(total_entries__gte=5, avg_entry_rating__gte=4.0)
```

Пример запроса

Задача: топ-5 блогов по среднему рейтингу записей за последние 6 месяцев + подгрузка авторов этих записей + сумма комментариев.

```
from django.db.models import Avg, Sum, Count, Q, Prefetch
from django.utils import timezone
from datetime import timedelta
six_months_ago = timezone.now() - timedelta(days=180)
top_blogs = (
    Blog.objects
    .annotate(
        # Средний рейтинг ТОЛЬКО свежих записей
        avg_rating=Avg('entry__rating', filter=Q(entry__pub_date__gte=six_months_ago)),
        # Сумма комментариев ТОЛЬКО свежих записей
        total_comments=Sum('entry__number_of_comments', filter=Q(entry__pub_date__gte=six_months_ago)),
        # Количество свежих записей (для статистики)
        entries_count=Count('entry__id', filter=Q(entry__pub_date__gte=six_months_ago))
    )
    .filter(avg_rating__isnull=False)  # Убираем блоги без свежих записей
    .order_by('-avg_rating')           # Сортировка по убыванию рейтинга
    .prefetch_related(
        # Оптимизированная подгрузка авторов ТОЛЬКО для свежих записей
        Prefetch(
            'entry_set',
            queryset=Entry.objects.filter(pub_date__gte=six_months_ago)
                                  .prefetch_related('authors')
                                  .order_by('-pub_date'),
            to_attr='recent_entries'   # Сохраняем в кастомный атрибут
        )
    )[:5]  # LIMIT 5
)
```