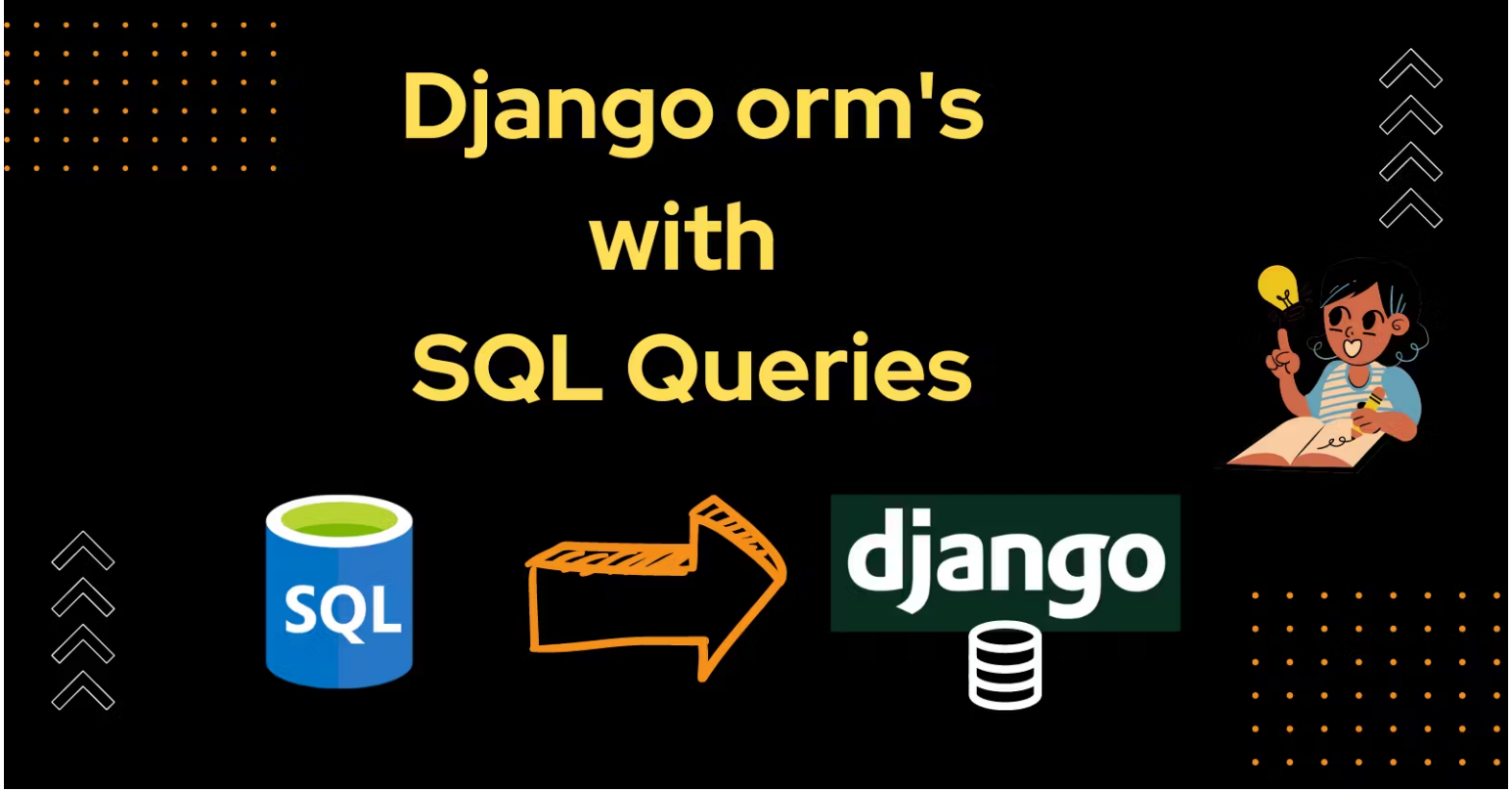

Mapping SQL queries with Django ORM | Cheatsheet

Understand common SQL queries and their mapping to Django ORM

Keshav Agarwal · Dec 1, 2022 · 6 min read

TABLE OF CONTENTS

Show more

Introduction

Today I will be describing **Django ORM** via **SQL queries**.

Many of you may know SQL queries but converting SQL queries to Django ORM is a basic challenge everyone faces. Let's learn and understand it properly today for better coding practices.

We will be covering the following:

> *We will start with the **Basic** and move to more **Advances** queries*

1. Create Table

Let us consider a simple base model for a Book with attributes title, pages, and price. If we want to **create a table** to store Book data, in SQL we need to run

```
CREATE TABLE Book (id int, title varchar(50), price int NOT NULL, pages in
```

The same table is modeled in Django as a class that inherits from the base Model class. The ORM creates the equivalent table under the hood.

```
class Book(models.Model):
    title = models.CharField(max_length=50, blank=True)
    price = models.IntegerField()
    pages = models.IntegerField()
```

The most used data types are:

SQL	Django

`INT`	`IntegerField()`
`VARCHAR(n)`	`CharField(max_length=n)`
`TEXT`	`TextField()`
`FLOAT(n)`	`FloatField()`
`DATE`	`DateField()`
`TIME`	`TimeField()`
`DATETIME`	`DateTimeField()`

> *Now let's learn how to work on the **Book Model** and its **relation with SQL queries***

2. Select the rows from the table

1. Fetch all rows
2. Fetch specific columns
3. Fetch distinct rows
4. Fetch specific rows
5. LIMIT & OFFSET keyword

(a) Fetch all data from Table

`SQL:`

```
SELECT * FROM Book;
```

`Django:`

```
books_data = Book.objects.all()
for book in books_data :
    print(book.title)
    print(book.price)
    print(book.pages)
```

(b) Fetch specific columns

`SQL:`

```
SELECT name, age FROM Book;
```

`Django:`

```
Person.objects.only('name', 'age')
```

(c) Fetch distinct rows

`SQL:`

```
SELECT DISTINCT name, age
FROM Person;
```

`Django:`

```
Person.objects.values('name', 'age').distinct()
```

(d) Fetch some specific number of rows

`SQL:`

```
SELECT * FROM Person LIMIT 10;
```

`Django:`

```
Person.objects.all()[:10]
```

(e) LIMIT & OFFSET keywords

- The **limit** keyword is used to limit the number of rows returned in a query result.
- The **OFFSET** value is also most often used together with the LIMIT keyword. The **OFFSET** value allows us to specify which row to start from retrieving data
- Note - Row count starts from 0

`SQL:`

```
SELECT *
FROM Person
OFFSET 5
LIMIT 5;
```

`Django:`

```
Person.objects.all()[5:10]
```

> *Now we will learn how to filter the data using the **WHERE clause***

3. Methods to Filter the rows from table

1. Comparison operators (>, < , >=, <=, !=)
2. BETWEEN clause
3. LIKE operator
4. IN operator
5. AND, OR, NOT operator

(a) Filter by a single column

`SQL:`

```
SELECT *
FROM Person
WHERE id = 1;
```

`Django:`

```
Person.objects.filter(id=1)
```

(b) Filter by comparison operators

`SQL:`

```
WHERE age > 18;
WHERE age >= 18;
WHERE age < 18;
```

```
WHERE age <= 18;
WHERE age != 18;
```

`Django:`

```
Person.objects.filter(age__gt=18)
Person.objects.filter(age__gte=18)
Person.objects.filter(age__lt=18)
Person.objects.filter(age__lte=18)
Person.objects.exclude(age=18)
```

(c) BETWEEN Clause

- Begin & End values are included

`SQL:`

```
SELECT *
FROM Person
WHERE age BETWEEN 10 AND 20;
```

`Django:`

```
Person.objects.filter(age__range=(10, 20))
```

(d) LIKE operator

`SQL:`

```
WHERE name like '%A%';
WHERE name like binary '%A%';
WHERE name like 'A%';
WHERE name like binary 'A%';
WHERE name like '%A';
WHERE name like binary '%A';
```

`Django:`

```
Person.objects.filter(name__icontains='A')
Person.objects.filter(name__contains='A')
Person.objects.filter(name__istartswith='A')
```

```
Person.objects.filter(name__startswith='A')
Person.objects.filter(name__iendswith='A')
Person.objects.filter(name__endswith='A')
```

(e) IN operator

`SQL:`

```
WHERE id in (1, 2);
```

`Django:`

```
Person.objects.filter(id__in=[1, 2])
```

(f) AND operator

`SQL:`

```
WHERE gender='male' AND age > 25;
```

`Django:`

```
Person.objects.filter(gender='male', age__gt=25)
```

(g) OR operator

`SQL:`

```
WHERE gender='male' OR age > 25;
```

`Django:`

```
from django.db.models import Q
Person.objects.filter(Q(gender='male') | Q(age__gt=25))
```

(h) NOT operator

`SQL:`

```
WHERE NOT gender='male';
```

`Django:`

```
Person.objects.exclude(gender='male')
```

(i) NULL checks

`SQL:`

```
WHERE age is NULL;
WHERE age is NOT NULL;
```

`Django:`

```
Person.objects.filter(age__isnull=True)
Person.objects.filter(age__isnull=False)

# Alternate approach
Person.objects.filter(age=None)
Person.objects.exclude(age=None)
```

4. Methods to Order the rows from Table

Now we will learn how to filter the data using the **ORDER BY keyword**

1. Ascending order
2. Descending order

(a) Ascending Order

`SQL:`

```
SELECT *
FROM Person
```

```
order by age;
```

`Django:`

```
Person.objects.order_by('age')
```

(b) Descending Order

`SQL:`

```
SELECT *
FROM Person
ORDER BY age DESC;
```

`Django:`

```
Person.objects.order_by('-age')
```

5. Method to Insert the rows in Table

`SQL:`

```
INSERT INTO Person
VALUES ('Jack', '23', 'male');
```

`Django:`

```
Person.objects.create(name='jack', age=23, gender='male)
```

6. Methods to Update the rows in Table

1. Update single row
2. Update multiple rows

(a) Update single row

`SQL:`

```
UPDATE Person
SET age = 20
WHERE id = 1;
```

`Django:`

```
person = Person.objects.get(id=1)
person.age = 20
person.save()
```

(b) Update multiple rows

`SQL:`

```
UPDATE Person
SET age = age * 1.5;
```

`Django:`

```
from django.db.models import F

Person.objects.update(age=F('age')*1.5)
```

7. Methods to Delete the rows from table

1. Delete all rows
2. Delete specific rows

(a) Delete all rows

`SQL:`

```
DELETE FROM Person;
```

`Django:`

```
Person.objects.all().delete()
```

(b) Delete specific rows

`SQL:`

```
DELETE FROM Person
WHERE age < 10;
```

`Django:`

```
Person.objects.filter(age__lt=10).delete()
```

8. Aggregation queries

1. MIN function
2. MAX function

3. AVG function
4. SUM function
5. COUNT function

(a) MIN Function

`SQL:`

```
SELECT MIN(age)
FROM Person;
```

`Django:`

```
>>> from django.db.models import Min
>>> Person.objects.all().aggregate(Min('age'))
{'age__min': 0}
```

(b) MAX Function

`SQL:`

```
SELECT MAX(age)
FROM Person;
```

`Django:`

```
>>> from django.db.models import Max
>>> Person.objects.all().aggregate(Max('age'))
{'age__max': 100}
```

(c) AVG Function

`SQL:`

```
SELECT AVG(age)
FROM Person;
```

`Django:`

```
>>> from django.db.models import Avg
>>> Person.objects.all().aggregate(Avg('age'))
{'age__avg': 50}
```

(d) SUM Function

`SQL:`

```
SELECT SUM(age)
FROM Person;
```

`Django:`

```
>>> from django.db.models import Sum
>>> Person.objects.all().aggregate(Sum('age'))
{'age__sum': 5050}
```

(e) COUNT Function

`SQL:`

```
SELECT COUNT(*)
FROM Person;
```

`Django:`

```
Person.objects.count()
```

9. Methods to Group By

1. Count of person by gender

(a) Count of Person by gender

`SQL:`

```
SELECT gender, COUNT(*) as count
FROM Person
GROUP BY gender;
```

`Django:`

```
Person.objects.values('gender').annotate(count=Count('gender'))
```

10. Convert HAVING in SQL to Django ORM

1. Count of Person by gender if number of person is greater than 1

(a) Count of Person by gender if number of person is greater than 1

`SQL:`

```
SELECT gender, COUNT('gender') as count
FROM Person
```

```
GROUP BY gender
HAVING count > 1;
```

`Django:`

```
Person.objects.annotate(count=Count('gender'))
.values('gender', 'count')
.filter(count__gt=1)
```

11. Convert JOINS in SQL to Django ORM

1. Fetch publisher name for a book
2. Fetch books which have specific publisher

Consider a foreign key relationship between books and publisher

```
class Publisher(models.Model):
    name = models.CharField(max_length=100)

class Book(models.Model):
    publisher = models.ForeignKey(Publisher, on_delete=models.CASCADE)
```

(a) Fetch publisher name for a book

`SQL:`

```
SELECT name
FROM Book
LEFT JOIN Publisher
ON Book.publisher_id = Publisher.id
WHERE Book.id=1;
```

`Django:`

```
book = Book.objects.select_related('publisher').get(id=1)
book.publisher.name
```

(b) Fetch books which have specific publisher

`SQL:`

```
SELECT *
FROM Book
WHERE Book.publisher_id = 1;
```

`Django:`

```
publisher = Publisher.objects.prefetch_related('book_set').get(id=1)
books = publisher.book_set.all()
```

I hope you liked😃 this post.

Please hit the Follow🤞 button below to read my future articles.

Hit a Clap👏 to cheer me up

Subscribe to my newsletter

Python Django Python 3 cheatsheet django orm

WRITTEN BY

Keshav Agarwal

Follow

As a coder and founder with 6 years of experience, I am passionate about sharing my knowledge and experience with others.

I believe in the power of mentorship and am dedicated to helping others learn and grow in their careers.

On my Hashnode blog, you'll find a wide range of technical content, including tutorials, best practices, and in-depth articles on the latest developments in software development.

Whether you're a beginner just starting out or an experienced developer looking to expand your skills, I aim to provide valuable insights and resources that will help you succeed.

My goal is to be a trusted voice in the developer community, and I welcome any opportunity to connect and collaborate.

ARTICLE SERIES

Django

1 **Mapping SQL queries with Django ORM | Cheatsheet**

Introduction Today I will be describing Django ORM via SQL queries. Many of you may know SQL queries...

2

Learn Django ORM Queries

Hi Guys! Today I will be describing Django ORM. We will learn ORM of some common SQL queries. Introd...