QuerySets and aggregations in Django

April 27, 2021 · 6 min read

Introduction

The object-relational mapper (ORM) in Django makes it easy for developers to be productive without prior working knowledge of databases and SQL. QuerySets represent a collection of objects from the database and can be constructed, filtered, sliced, or generally passed around without actually hitting the database. No database activity occurs until we do something to evaluate the QuerySet. In this guide, you will learn about how to perform these queries, both basic and advanced.

Throughout the guide, we will refer the `django.contrib.auth.models.User` model. You can insert multiple users into this model to test different QuerySets discussed in the following guide.

Moreover, we will be using the Django shell for running and testing the queries. You can start the Django shell with the following:

```
python manage.py shell
```

Basic Queries

Let’s start with some basic QuerySet operations.

Retrieving single objects

For cases in which you know there is only a single object that matches the query, you can use the `get()` method, which will return the object. Unlike `filter`, which always returns the `QuerySet`:

```
>>> user_id_1 = User.objects.get(id=1)
```

Note that if no results are found, it will raise a `DoesNotExist` exception, so better to use it in the try-except block:

```
try:
user_id_1 = User.objects.get(id=1)
except User.DoesNotExist:
print("User with id does not exists")
```

Getting an object from the QuerySet

There are two options for getting an object from the QuerySet.

The first is using `first()` and `last()`. `First()` returns the first object matched to the QuerySet, and `last()` returns the last object matched to the QuerySet:

```
from django.contrib.auth.models import User

>>> User.objects.filter(is_active=True).first()

>>> User.objects.filter(is_active=True).last()
```

The above query will return the first and last object matched with the Queryset.

The second option is `latest()` and `earliest()`. `Latest()` returns the latest object in the table based on the given fields, and `earliest` returns the earliest object in the table based on given fields:

```
from django.contrib.auth.models import User

>>> User.objects.latest('date_joined')

>>> User.objects.earliest('date_joined')
```

Field lookups

Field lookups deal with how you specify the SQL `WHERE` clause. Basic lookup keyword arguments take the form `field__lookuptype=value` . For example:

```
from datetime import datetime
## Get all users whose date_joined is less than today's date.
>>> User.objects.filter(date_joined__lte=datetime.today())
```

Searching the specific string (case sensitive):

```
## Get all user whose username string contains "user"
>>> User.objects.filter(username__contains = "user")
```

Or case insensitive:

```
## Get all user whose username string contains "user" (case insensitive)
>>> User.objects.filter(username__icontains = "user")
```

Or, starts-with and ends-with search:

```
## Get all user whose username string starts with "user"
>>> User.objects.filter(username__startswith = "user")
## Get all user whose username string ends with "user"
>>> User.objects.filter(username__endswith = "user")
```

You could also use case-insensitive versions called `istartswith` and `iendswith` .

Ordering QuerySets

After filtering the QuerySet, you can order it ascending or descending based on the given field(s).

The below query will first filter the users based on `is_active` , then by username in ascending order, and finally by `date_joined` in descending order. Note that `-` indicates the descending order of `date_joined` :

```
from django.contrib.auth.models import User

>>> User.objects.filter(is_active=True).order_by('username', '-date_joined')
```

Chaining filters

Django gives the option to add several filters to chain refinements together:

```
import datetime
from django.contrib.auth.models import User

>>> User.objects.filter(
... username__startswith='user'
... ).filter(
... date_joined__gte=datetime.date.today()
... ).exclude(
... is_active=False
... )
```

The above query initially takes all users, adds two filters, and excludes one. The final result is a QuerySet containing all users whose `username` starts with `user`, their `date_joined` being greater or equal to today's date, and finally, excludes the inactive users.

Advanced queries

Now, that you understand the basic QuerySet operations, let's now jump to advanced queries and QuerySet operations.

Set operations

`Union()` uses SQL `UNION` operator to combine the results of two or more QuerySets:

```
>>> qs1.union(qs2, qs3, ...)
```

`Intersection()` uses the SQL `INTERSECTION` operator to find common(shared) results of two or more QuerySets:

```
>>> qs1.intersection(qs2, qs3, ...)
```

`Difference()` uses the SQL `EXCEPT` operator to find elements present in the QuerySet but not in some other QuerySets:

```
>>> qs1.difference(qs2, qs3, ...)
```

Q objects

A `Q()` object represents an SQL condition that can be used in database-related operations. If you want to execute complex queries that contain `OR`, `AND`, and `NOT` statements, you can use `Q()` objects:

```
>>> from django.db.models import Q

>>> Q(username__startswith='user')
<Q: (AND: ('username__startswith', 'user'))>
```

For example, let's find all users who are either staff or superusers:

```
>>> from django.contrib.auth.models import User

>>> User.objects.filter(Q(is_staff=True) | Q(is_superuser=True))
```

Similarly, you could use `AND` and `NOT`. In the below query, it finds all the users who are staff and whose usernames do not start with `user`:

```
>>> User.objects.filter(Q(is_staff=True) & ~Q(username__startswith='user'))
```

F objects

The `F()` object represents the value of a model field or annotated column. It makes it possible to refer to model field values and perform database operations using them without actually having to pull them out of the database into Python memory.

Let's take an example of incrementing a hit count by one with the `HitCount` model of `id=1`.
Normally, one obvious way is to save it in memory, increment the count, and then save it:

```
site = HitCount.objects.get(id=1)
site.hits += 1
site.save()
```

The other way we can deal with this entirely by the database is by introducing the `F()` objects. When Django encounters an instance of `F()`, it overrides the standard Python operators to create an encapsulated SQL expression:

```
from django.db.models import F

site = HitCount.objects.get(id=1)
site.hits = F('hits') + 1
site.save()
```

`F()` offers performance advantages by:

- Getting the database, rather than Python, to perform operations
- Reducing the number of queries some operations require

Performing raw SQL queries

Django provides two ways of performing the raw SQL queries using `raw()` and `connection.cursor()`.

For clarity, let's take a basic query of fetching the non-staff users:

```
from django.contrib.auth.models import User

User.objects.filter(is_staff = False)
```

Executing raw queries

`Raw()` takes a raw SQL query, executes it, and returns a `RawQuerySet` instance, which can be iterated over like a normal QuerySet to provide object instances:

```
query = "select * from auth_user where is_staff=False;"
results = User.objects.raw(query)
for result in results:
print(result)
```

Executing the custom SQL directly

Sometimes even `raw` isn't enough; you might need to perform queries that don't map cleanly to models, or directly execute `UPDATE` , `INSERT` , or `DELETE` queries. In these cases, you can always access the database directly, routing around the model layer entirely.

For example, you can run the above SQL query using the cursor as demonstrated below:

```
from django.db import connection

query = "select * from auth_user where is_staff=False;"
with connection.cursor() as cursor:
cursor.execute(query)
print(cursor.fetchall())
```

Refer more on this topic from Django's documentation here.

Getting raw SQL for a given QuerySet

To get the raw SQL query from a Django QuerySet, the `.query` attribute can be used. This will return the `django.db.models.sql.query.Query` object, which then can be converted to a string using `__str__()` :

```
>>> queryset = MyModel.objects.all()
>>> queryset.query.__str__()
from django.contrib.auth.models import User

>>> queryset = User.objects.all()
>>> queryset.query
<django.db.models.sql.query.Query at 0x1ff0dcf7b08>

>>> queryset.query.__str__()
'SELECT "auth_user"."id", "auth_user"."password", "auth_user"."last_login",
"auth_user"."is_superuser", "auth_user"."username", "auth_user"."first_name",
"auth_user"."last_name", "auth_user"."email", "auth_user"."is_staff",
"auth_user"."is_active", "auth_user"."date_joined" FROM "auth_user"'
```

Aggregation

Grouping by queries is fairly common SQL operations, and sometimes becomes a source of confusion when it comes to ORM. In this section, we will dive into applying `GROUP BY` and aggregations.

Basic `GROUP BY` and aggregations

Let's start with basic count operations, which will return the `dict` containing the count of users:

```
>>> User.objects.aggregate(total_users=Count('id'))
```

Using annotate

`Aggregate` is used to the aggregate whole table. Most of the time we want to apply the aggregations to groups of rows, and for that, `annotate` can be used.

Let's look at an example to group users based on `is_staff` :

```
>>> User.objects.values("is_staff").annotate(user_count=Count('*')
```

To perform group by in ORM style, we have to use the two methods `values` and `annotate` as follows:

- `values(<col>)` : Mention the fields for what to group by
- `annotate(<aggr function>)` : Mention what to aggregate using functions such as `SUM` , `COUNT` , `MAX` , `MIN` , and `AVG`

Multiple aggregations and fields

For multiple aggregations, we need to add multiple fields by which you want to group. In the below example, we have executed a query group by columns (`is_active` , `is_staff`):

```
>>> User.objects.values("is_active", "is_staff").annotate(user_count =
Count("*"))
```

`HAVING` clause

The `HAVING` clause is used to filter groups. In the below query, I have filtered the group which has a count greater than one:

```
objects.values("is_staff").annotate(user_count=Count("*")).filter(user_count__gt
```

The equivalent SQL query is:

```
SELECT is_staff, COUNT(*) AS user_count
FROM auth_user
GROUP BY is_staff
HAVING COUNT(*) > 1;
```

Conclusion

In this guide, we have discussed various methods of QuerySets and how to work with different queries. With some care and understanding of the simple concepts behind Django's QuerySets, you can improve your code and become a better Django developer. You can always refer to the Queryset documentation and Aggregation documentation for further study.

Gaurav is a data scientist with a strong background in computer science and mathematics. As a developer, he works with Python, Java, Django, HTML, Struts, Hibernate, Vaadin, web scraping, Angular, and React.

#django #python