Theory: Aggregations and Ordering

33 minutes 0 / 5 problems solved Skip this topic Start practicing

240 users solved this topic. Latest completion was about 9 hours ago.

Filtering data is the same as asking which objects with given features we have. Queries to the database can answer some other questions. For example, how many objects do we have? How much money is there on all bank accounts? What is the maximum height of all towers in Spain? We can also gather our data in groups and *aggregate* the values for each of them.

Another operation which may be really helpful is *sorting* data by its attributes. Let's see how it works in Django ORM.

§1. Ordering

Everybody wants to know what will happen in the future. Even the most sophisticated level of programming can't give you that – but come to think about it, weather forecasts are a form of fortune telling. The methods are not quite accurate yet, but we may rely on historical data and statistics. The only model in our application is called `DayWeather`:

```
from django.db import models


class DayWeather(models.Model):
    date = models.DateField()
    precipitation = models.FloatField()
    temperature = models.FloatField()
    was_raining = models.BooleanField()
```

Are you curious to find the 3 coldest days in our database? Let's look for an answer:

```
top_three_coldest_days = DayWeather.objects.order_by('temperature')[:3]
```

Remember that ordering works like sorting: the first value is the smallest and the last is the biggest. We call the `order_by` method of Object Manager with *temperature* as a parameter and it returns data sorted by temperature field.

This method may be applied for Object Manager or QuerySet; you may filter data and then order it or even make aggregation operations and sort the total result.

How about getting the top 3 hottest days from the database?

```
top_three_hottest_days = DayWeather.objects.order_by('-temperature')[:3]
```

All that changes is adding the minus to our parameter. The minus sign means the reversed order. In our case, we get the biggest values of temperature first.

§2. Aggregations

The simple rule of weather forecasts is that the weather tomorrow is going to be relatively similar to the weather today. We also know that the weather is seasonal. Say, if we want to know how many days will be rainy and gloomy next month, we should look at how many days it rained in the same month last year. Assume that we have variables `last_year` and `next_month`:

```
raining_days = DayWeather.objects.filter(
    date__year=last_year, date__month=next_month, was_raining=True
)

raining_days_forecast = raining_days.count()
```

Current topic:

Aggregations and Ordering

Topic depends on:

✓ Slicing

✓ Operations with list Stage 3 6★

✗ Field lookups

Table of contents:

We make a QuerySet and call the method `count` on it. We think it is a good approximation to count the number of raining days in the same month last year and base our forecast on this number.

We may also wonder about the average temperature next week. For this prediction we analyze the temperature for the past week:

```
from datetime import date, timedelta
from django.db.models import Avg


query = DayWeather.objects.filter(date__gt=date.today() - timedelta(days=7))

average_temperature = query.aggregate(average=Avg('temperature'))
['average']
```

We call the `aggregate` method and pass any custom name as a parameter. The value of this parameter is a special function `Avg`, and the parameter of the `Avg` function is the name of the field we want to process.

The result of this function is Python's dictionary `{'average': ...}`. We get the value of the average temperature by the custom name we chose early in `aggregate` method.

It almost looks as programmers may indeed be almighty, acing everything from Hello, World to predicting this world's future – at least to an extent.

§3. Group By Aggregations

We look through Django aggregation functions and find `Avg`, `Count`, `Max`, `Min`, `StdDev`, `Sum`, `Variance`. We can apply any of these functions to the numerical field values of the QuerySet.

The other task is to predict the total precipitation for each month for a whole year ahead. Should we create twelve QuerySets and process them one by one? Well, this is one way to do it; the other is to group values by month:

```
from django.db.models import Sum

precipitation = DayWeather.objects.filter(date__year=last_year) \
                          .values('date__month') \
                          .annotate(sum=Sum('precipitation'))

# precipitation is <Queryset [{'date__month': 1, 'sum': ...}, ...]>
```

We make a query and group our values by the month calling `values` method and passing a field or a field lookup to it. Then we call the `annotate` method; the syntax rules for it are the same as for `aggregate` in the previous example.

The result this time is a QuerySet consisting of customized objects in the form of dictionaries. We can access each object by index or convert the QuerySet to Python collection and work with it as we would with any other collection.

§4. Count Function vs Count Method

The last prediction we want to make is the number of warm days with the outside temperature greater than or equal to 20 degrees Celsius per week. We again look at the values of the last year for each week:

```
from django.db.models import Count

warm_days = DayWeather.objects.filter(date__year=last_year) \
                      .filter(temperature__gte=20) \
                      .values('date__week') \
                      .annotate(count=Count('date'))

# warm_days is <Queryset [{'date__week': 1, 'count': ...}, ...]>
```

The call is similar to the previous one, but this time we pass the function `Count` to the `annotate` method.

There is a difference between the Count function and the count method of a QuerySet. The Count function returns statistics for each annotated value; the method returns the number of elements in the whole QuerySet.

We made some simple assumptions about seasons and cycles and released our forecasts. Not all of them were good, but remember: if at first you don't succeed, then try, try and try again.