How to Use Materialized View in Django

Django is one of the most popular Python Web framework. It is designed based on the "model-template-view" pattern. In Django, the Model is acting as the interface to the data (Database). As each model is associated with each physical table in database, I start wondering if it's possible to make ORM deal with more advanced database feature, that is view and materialized view. The result turns out to be more than great!

View and materialized view are results of lookup SQL query. The main difference between them is that materialized view stores result as an actual table while view does not. From technical perspective, view and materialized view are good for repetitive query. They enhance query speed. From business perspective, they can be used to declare data scope for the usages of different teams/departments. The question is how we leverage this feature in Django?

Here are conclusions of my research -- ORM works well with materialized view but view. It makes sense, does it? Because materialized view is a physical table. ORM should be able to associate it. Regarding view, ORM has a similar mechanism called "manager" on which I will write another article to walk you through in the future. Let's continue our tutorial on materialized view.

Dependencies

1. Python 3.7
2. Django 2.2.x
3. Psycopg2-binary 2.8.6

In this tutorial, I am going to use PostgreSQL database. Definitely choose whatever relational database you would like to use. In order to let Python "talk" to PostgreSQL database, we need to pip install **psycopg2-binary**. In addition, you may need to set up a local PostgreSQL database on your own machine so that models in your Django project can connect.

Step 0. Create local database

Download and install PostgreSQL engine as below. Follow the steps. You should be able to create a local database.

https://www.postgresql.org/download/

Regarding database client, I would like to use DBeaver community version. There are many options. It's up to you.

https://dbeaver.io/

Step 1. Build model to create an origin table

In models.py in **polls** app, declare a **Question** model as below.

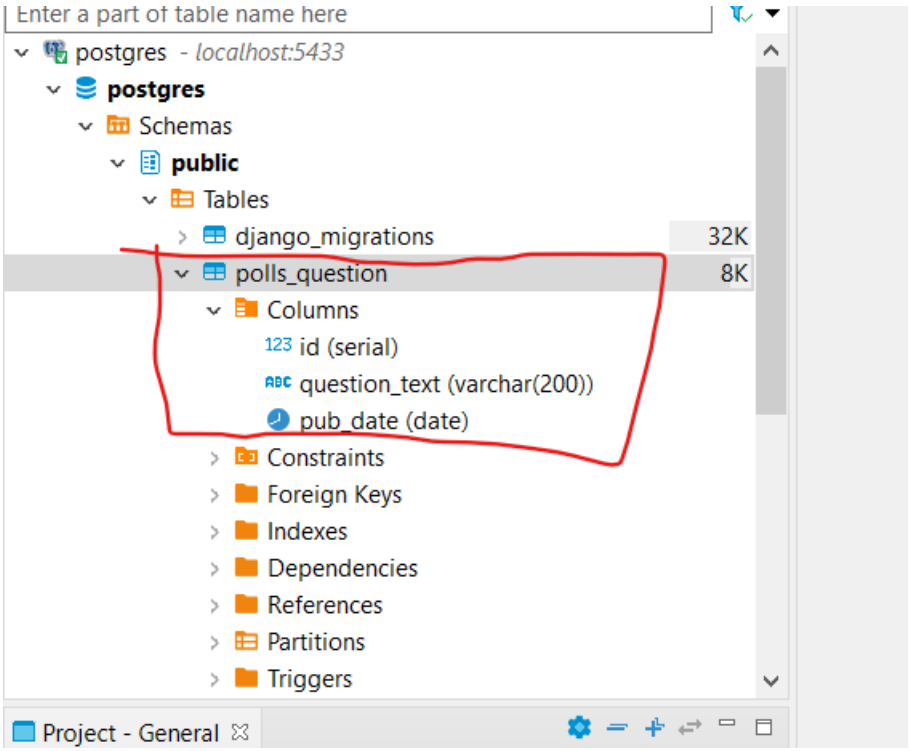
```
from django.db import models


class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateField(null=True, default=None)
```

Run migration commands as below. Await first command making impact, before we run second command.

```
python manage.py makemigration polls
python manage.py migrate polls
```

That should create a **polls_question** table in database. In my case, my local database name is **postgres**. This table is located under **public** schema. Columns are illustrated as below.



Let's populate some data into table.

Run command below that should bring us to a Django console.

```
python manage.py shell
```

Copy and paste following codes in console. Hit **Enter**. It should insert 1000 rows. Now we have enough data to test!

```
from polls.models import Question
import string
import random
import datetime


N = 10
questions = []
for each in range(1000):
    res = ''.join(random.choices(string.ascii_uppercase + string.digits, k=N))
    syear = random.randint(2018, 2021)
    smonth = random.randint(1, 12)
    sdate = random.randint(1, 20)
    pub_date = datetime.date(year=syear, month=smonth, day=sdate)
    questions.append(Question(question_text=res, pub_date=pub_date))

Question.objects.bulk_create(questions)
```

Step 2. Create lookup query on origin table

In **views.py** of **polls** app, add a lookup query by selecting all data in **polls_question** table.

```
from django.shortcuts import render
from polls.models import Question
import datetime


def index(request):
    questions = Question.objects.all()
    print('Origin table returns {} questions'.format(questions.count()))

    return render(request, 'polls/index.html')
```

Launch project. Open up http://127.0.0.1:8000/polls/ in browser. We should see a print as below in console. It returns 1000 rows. It's what we would expect.

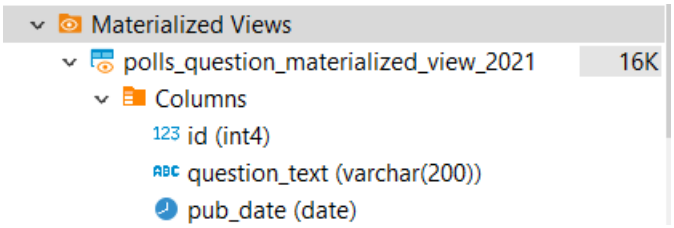
```
Quit the server with CTRL-BREAK.
Origin table returns 1000 questions
[09/May/2021 16:33:17] "GET /polls/ HTTP/1.1" 200 10477
```

Step 3. Create a materialized view in database

In PostgreSQL console, run following SQL command to generate a materialized view that only contains data whose pub_date is in or after year 2021.

```
create materialized view polls_question_materialized_view_2021 as
select * from polls_question
where pub_date>='2021-01-01'
```

We should see it under Materialized Views category as below.



Step 4. Build and associate model with materialized view

In **models.py** in **polls** app, create a model **Question2021** as below.

```
class Question2021(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateField(null=True, default=None)

    class Meta:
        managed = False
        db_table = 'polls_question_materialized_view_2021'
```

It's important to declare db_table explicitly so that Django can make connection between model and materialized view. Please do not run migration. Because materialized view is already created manually through SQL command. Plus, we need to make sure **managed** attribute is False. It makes sure Django does not create table by mistake when we run migrations on **polls** app in the future.

Step 5. Create lookup query on materialized model

Add lookup query by selecting all data on **Question2021** model and print number of returns in console. Codes should look like below.

```
from django.shortcuts import render
from polls.models import Question, Question2021


def index(request):
    questions = Question.objects.all()
    print('Origin table returns {} questions'.format(questions.count()))

    questions_2021 = Question2021.objects.all()
    print('Materialized view returns {} questions.'.format(questions_2021.count()))

    return render(request, 'polls/index.html')
```

Launch project. Open up http://127.0.0.1:8000/polls/ in browser. We should see two prints as below in console. Materialized view returns less rows than origin table returns. It works as expected!

```
Origin table returns 1000 questions
Materialized view returns 268 questions.
```

Conclusion

In this article, we've created materialized view in database and associate ORM model with it. It's quite straightforward, is it? For architecture perspective, hope it gives you a better idea how to manage your data queries engine. Thanks for reading. Stay tuned.

If you are interested, I've uploaded this project here.

https://github.com/slow999/DjangoAndMaterializedView