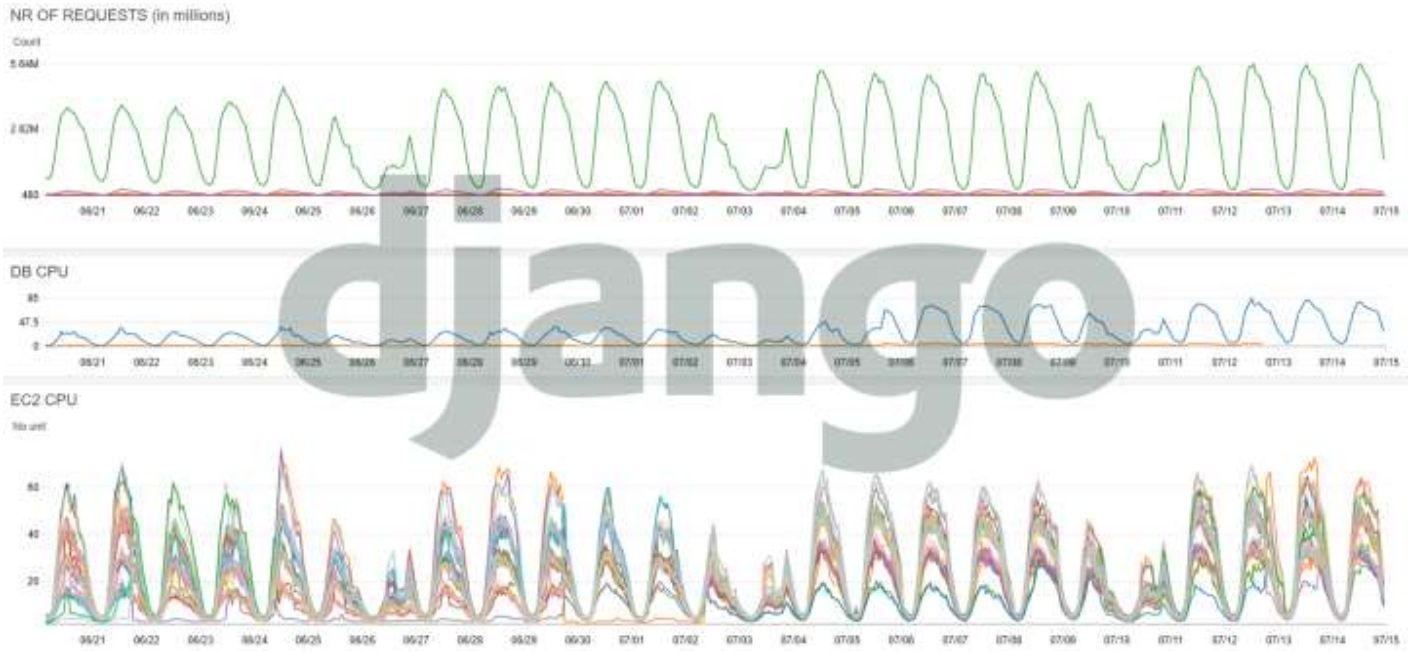
Nicolae Godina Follow
Jul 21, 2020 · 6 min read

How Django can handle 100 millions of requests per day

Today I'll write about Django, my loyal framework-companion for the last five years. This one helped me succeed in developing high-load solutions used by millions of users to date.

It is true, Python is not a very "fast" programming language, however, it's simple, loved, and convenient. In terms of execution performance, it might not be as fast as Go or NodeJs, but this becomes irrelevant when considering modern infrastructures and modular development.

As I've been boiling in the Django kettle for years now, I've reached several insightful conclusions I am about to share with you.



#1. Infrastructure Matters

Despite application performance, the first thing you need is an infrastructure that allows you to scale when the app reaches its limit and Django can scale-up easily if you follow these rules:

- Split your application into microservices, but consider the volume of data transferred between them, especially since data redundancy and frequent synchronizations lead to increased server resources and communication, thus a higher cost;
- Use Docker containers to place your code in production;

- Docker Containerisation is not enough, hence use Kubernetes to orchestrate containers and control the number of replicas;
- Design your infrastructure with maintenance in mind: a proper one will allow you to upgrade or downgrade server resources without experiencing service downtime;
- Collect and monitor metrics that matter: number of requests per microservice and each endpoint, CPU usage on each pod, CPU usage on k8s nodes, inbound and outbound traffic, CPU at database and storage usage — this will allow your to spot and fix issues on the fly, stepping-up from traditional troubleshooting to proactive maintenance;

#2. The database is your most likely culprit

Whatever speed you gain via code execution you'll most probably lose at the database end. Particularly, the response speed of an endpoint depends on how fast your database query is processed, hence, you should check on the following:

- Choose your DB engine wisely and focus on its performance — my preference is PostgreSQL because it has earned a strong reputation for its proven architecture, reliability, data integrity and performance;
- When deploying your data layer focus primarily on fast storage and CPU. You need to choose the best option of IOPS and the number of available CPU cores — no question about it;
- Check that you have created all the necessary Indexes for all queries;
- Remember that too many indexes are bad — delete unused or redundant ones: each created index might increase search metrics on that column (SELECT) but will reduce write speeds (INSERT, UPDATE). Django can create some indexes that repeat themself, hence, you have to check and delete them.

#3. Enable debug logs in Django ORM

During development, it's paramount to keep an eye on what query generates the ORM and the response speed. When you create an endpoint you have to make sure that its response is under 100ms, this is why you have to make sure queries are executed up to 20ms.

To activate the logs and see how long each query runs make use of these lines of code within settings.py:

```
LOGGING = {
    'version': 1,
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
        },
    },
    'loggers': {
        'django.db.backends': {
            'level': 'DEBUG',
        },
    },
    'root': {
        'handlers': ['console'],
```

```
        }
    }
```

And after the restart, you should see the queries in the following format:

```
(0.080) SELECT "django_migrations"."app", "django_migrations"."name" FROM "django_migrations"; args=()
```

The first number is the query execution time

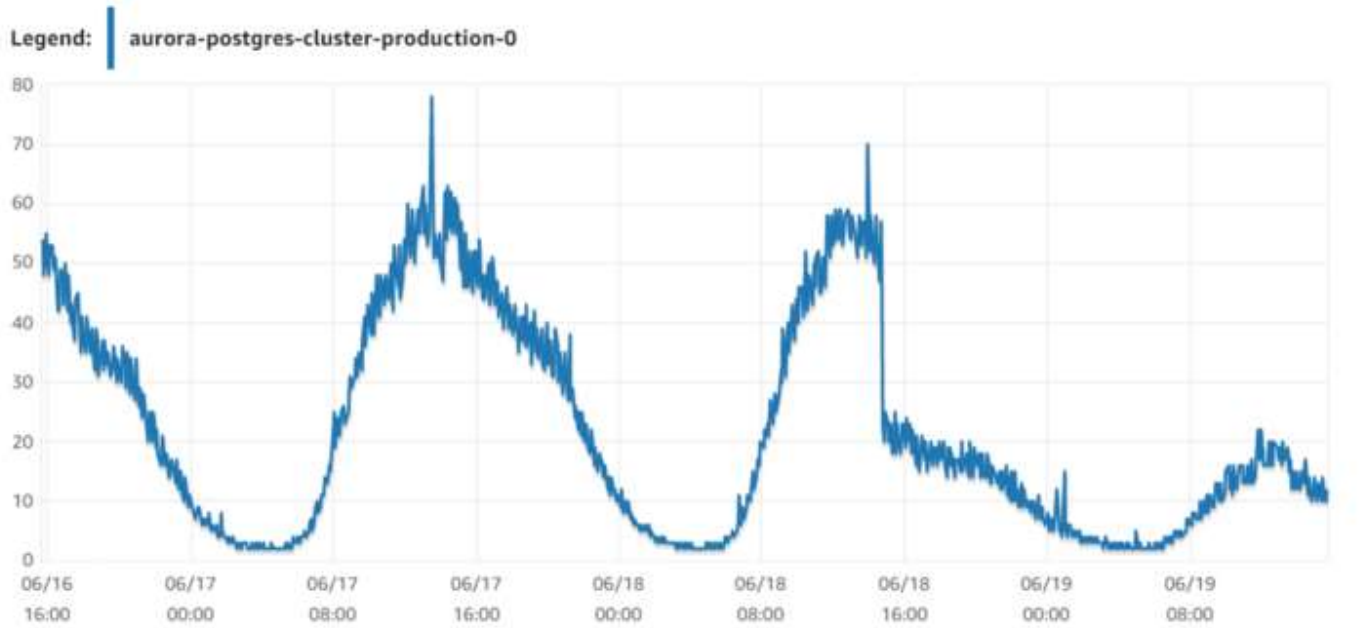
If your choice is PostgreSQL, I recommend using pghero, a performance dashboard to view Slow queries and Duplicated indexes.

#4. Enable persistent connections

If the application needs to process a large number of requests, enable maintaining persistent connections to the database. Django closes the connection by default at the end of each request and persistent connections avoid overloading the database for each request.

These connections are controlled by the **CONN_MAX_AGE** param, a metric that defines the maximum lifetime of a connection. Set a suitable value depending on your request volume at the applications' end. I usually limit it to expire in 5 minutes. Make sure that the database is not limited in matters of concurrent connection figure, usually, the default number of connections is 100 — which is not nearly enough in most high-load scenarios.

For instance, in one of my production projects, after adjusting this parameter from 0 to 300 seconds, I reduced the DB load in half. I made use of AWS Aurora DB engine with db.r5.8xlarge, downgrading it to db.r5.4xlarge to cut on expenses while maintaining performance.



#5. Deactivate unused apps and middlewares

By default, the framework has several applications activated that can be useless, especially if you use Django as a REST API. Consider **Sessions** and **Messages** in this scenario — these are useless and would just waste resources and reduce processing speeds. The fewer middleware you have declared, the faster each request will be processed.

```
25  25  # Application definition
26  26
27  27  INSTALLED_APPS = [
28      -     'django.contrib.admin',
29  28        'django.contrib.auth',
30  29        'django.contrib.contenttypes',
31      -     'django.contrib.sessions',
32      -     'django.contrib.messages',
33  30        'django.contrib.staticfiles',
34  31        'raven.contrib.django.raven_compat',
35  32        'rest_framework',
... ...   @@ -41,11 +38,8 @@ INSTALLED_APPS = [
41  38    ]
42  39    MIDDLEWARE = [
43  40        'django.middleware.security.SecurityMiddleware',
44      -     'django.contrib.sessions.middleware.SessionMiddleware',
45  41        'django.middleware.common.CommonMiddleware',
46  42        'django.middleware.csrf.CsrfViewMiddleware',
47      -     'django.contrib.auth.middleware.AuthenticationMiddleware',
48      -     'django.contrib.messages.middleware.MessageMiddleware',
49  43        'corsheaders.middleware.CorsMiddleware',
50  44        'django.middleware.clickjacking.XFrameOptionsMiddleware',
51  45        'django.middleware.locale.LocaleMiddleware',
```

#6. Use bulk query

Use bulk queries to efficiently query large data sets and reduce the number of database requests. Django ORM can perform several inserts or update operations in a single SQL query.

If you're planning on inserting more than 5000 objects, specify **batch_size.** Large batches will also decrease processing times and high memory consumption in Python, hence, you must find an optimal number of elements depending on the size of the object.

```python
bulk_inserts, bulk_updates, bulk_deletes = [], [], []

for favorite_number in sent_favorites - existing_favorites:
    bulk_inserts.append(Favorite(user=user, **favorites.get(favorite_number)))

for favorite_number in existing_favorites - sent_favorites:
    bulk_deletes.append(favorite_number)

for favorite_number in sent_favorites.intersection(existing_favorites):
    favorite = current_favorites.get(favorite_number)
    favorite.order = favorites.get(favorite_number).get('order')
    bulk_updates.append(favorite)

if bulk_inserts:
    Favorite.objects.bulk_create(bulk_inserts, batch_size=1000)

if bulk_updates:
    Favorite.objects.bulk_update(bulk_updates, ['order'], batch_size=1000)

if bulk_deletes:
    Favorite.objects.filter(user=user, phone_number__in=bulk_deletes).delete()
```

Example of bulk query in Django

#7. Reduce the number of select operations with select_related

If you have two related models and you need to pull specific properties from both of them, pre-select required entities via JOIN.

Here's an unfortunate example that illustrates generating 11 useless queries to the database:

```python
contacts = Contact.objects.filter(phone_number=request.user.phone_number)[:10]

for contact in contacts:
    print(contact.name, contact.user.first_name)
```

On the other hand, here's the right way o do it, generating only one query:

```python
contacts = Contact.objects.filter(phone_number=request.user.phone_number).select_related('user')[:10]

for contact in contacts:
    print(contact.name, contact.user.first_name)
```

Using **select_related** depends on table sizes since ORM generates the JOIN SQL query. To achieve optimization the WHERE condition must return a small number of rows.

#8. Reduce data transfer between your data and application layer

Focus on essential information from the database. Selecting unnecessary columns increases response times from the database resulting in data transfer costs.

Django ORM has a **.only()** QuerySet function for selecting specific fields or you can call **.defer()** to tell Django not to retrieve some fields from the database:

```python
contacts = Contact.objects.filter(phone_number=request.user.phone_number).only('name', 'email')

for contact in contacts:
    print(contact.name, contact.email)
```

Selecting a name and an email from a table

#9. Reduce data transfer between your API and clients

Similar to a strict selection from the database, it's paramount to return to essential information from the API. Because JSON is not the most efficient way to send data, you need to reduce the size by excluding fields that are not used by the app client.

As an illustration: the size of an answer from a specific endpoint is 1Kb, but if it is called 1 *million* times a day, 1 GB of data will be transferred daily, which means 30 GB per month, a pretty steep price to pay in resource usage.

Conclusion

Of course, it is easy to blame Django or Python, however, as my colleagues say: "Don't blame the piano — blame the pianist".

When developing a high-load project on Django, every little count. Issues thinner than a hair, multiplied by millions result in a pretty furry situation and you'll have to do all the trimming.

Any extra milliseconds multiplied by millions of requests can lead to excessive consumption of resources. If the application is optimized or properly built, increasing hardware resources does not save the day.

Take a page from Instagram, Pinterest or Disqus — they started with Django as it and took it to the next level. Sure, it might not be the same framework anymore, however, when commonsense is applied at the core, there are only fruits to bear.

Write code efficiently, reuse, use bulk, monitor, measure and optimize. Will get back to you soon.