Django Optimization

Ikhsan N. Rosyidin
@essanpupil

What are we optimizing for?

1. Fewer processing time
2. Less database query
3. Lower memory consumption
4. Easier maintenance & development

*"Improvements in one area will often bring about improved performance in another, but **not always**; sometimes one can even be at the expense of another."*

Know Your Profile

1. Django debug toolbar
2. Silk
3. django-querycount
4. Newrelic
5. Datadog
6. Elasticsearch APM

We must know our condition before doing optimization.

Use widely available tools to measure your current condition.

Database Indexing

```
class Student(models.Model):
    address = models.CharField(max_length=255)
    name = models.CharField(max_length=128, db_index=True)
    class Meta:
        Indexes = [
            models.Index(fields=[‘name’], name=’name_idx’)
         ]
# Use class Meta or db_index, don’t need to use both technic for same field
```

Consider adding indexes to fields that you frequently query using filter(), exclude(), & order_by().

Django ORM Optimization

Queryset

To avoid performance problems, it is important to understand:

1. queryset evaluation.
2. queryset data caching.

QuerySets Evaluation

```
q = Entry.objects.filter(headline__startswith="What")              # Not evaluated
q = q.filter(pub_date__lte=datetime.date.today())                  # Not evaluated
q = q.exclude(body_text__icontains="food")                         # Not evaluated

# Queryset will be evaluated when it is used for:

for person in Person.objects.all():                                # Iteration
     if person.age >= 10:
          print(“You are not a kid”)

first_person = Person.objects.all()[0]                             # Slicing/Indexing
json_persons = pickle.dumps(Person.objects.all())                  # Pickling (i.e. serialization)
print(q)                                                           # Function evaluation
list_person = [person for person in Person.objects.all()]          # List comprehensions

if person in Person.objects.all():                                 # `in` checks
     print(“You are included”)
```

QuerySets data Caching

```
# Not cached

# QuerySet evaluated and cached
print([p.name for p in Person.objects.all()])

# New QuerySet is evaluated and cached
print([p.name for p in Person.objects.all()])

# Slicing/indexing unevaluated QuerySets
queryset = Person.objects.all()

# Queries the database
print(queryset[0])

# Queries the database again
print(queryset[0])

# Printing
print(Person.objects.all())
```

```
# Cached
queryset = Person.objects.all()

# QuerySet evaluated and cached
print([p.name for p in queryset])

# Cached results are used
print([p.name for p in queryset])

# Slicing/indexing evaluated QuerySets
queryset = Person.objects.all()

# Queryset evaluated and cached
list(queryset)

print(queryset[0])  # Cache used
print(queryset[0])  # Cache used
```

Too Much Cache Will Kill You.

When you have a lot of objects, the caching behavior of the QuerySet can cause a large amount of memory to be used. In this case, iterator() may help.

```
# Save memory by not caching
# anything

for person in
Person.objects.iterator():
    # Some logic
```

Retrieve Everything at Once

Hitting the database multiple times for different parts of a single ‘set’ of data that you will need all parts of is, in general, less efficient than retrieving it all in one query. Use:

1. select_related()
2. prefetch_related()
3. prefetch_related_objects()

```
# Foreign Key & One to One
Child.objects.select_related(‘parent’).all()

# Many to Many & Many to One
Parent.objects.prefetch_related(‘children’).all()
```

Don't retrieve things you don't need

1. Values() & values_list() used to get value of specific field in list format.
2. Defer() & only() used to avoid unused querying field.
3. count() & exists() used if only care about number of rows.
4. update() & delete() used when we don't need object value but want to delete & update it.

HTTP Performance

1. Middleware
    a. ConditionalGetMiddleware, adds support for modern browsers to conditionally GET responses based on the ETag and Last-Modified headers.
    b. GZipMiddleware, compresses responses for all modern browsers, saving bandwidth and transfer time. **Warning:** Compression techniques used on a website increase attack possibility. Read detail possible attack in http://breachattack.com/
2. Sessions
    a. Use cached sessions instead of default database sessions.
3. Static Files
    a. ManifestStaticFilesStorage, appends a content-dependent tag to the filenames of static files to make it safe for browsers to cache them long-term.
    b. Minification, several third-party Django tools and packages provide the ability to “minify” HTML, CSS, and JavaScript. They remove unnecessary whitespace, newlines, and comments, and shorten variable names, and thus reduce the size of the documents that your site publishes.

Template Performance

1. Using {% block %} is faster than using {% include %}
2. Heavily-fragmented templates, assembled from many small pieces, can affect performance.
3. cached template loader often improves performance drastically, as it avoids compiling each template every time it needs to be rendered.
4. Use alternative template language, like; jinja2, mako, etc.

Dev is not Prod

1. Hardware specs.
2. Software configuration.
3. Database size.
4. Request numbers.

Setup staging environment, make it similar with production environment as much as possible.

Final Test

1. Server Side
    a. Locust.io
    b. Httperf
    c. jMeter
2. Client side
    a. Sitespeed.io
    b. webpagetest.org

References

1. https://docs.djangoproject.com/en/2.2/topics/performance/
2. https://techbeacon.com/app-dev-testing/web-performance-testing-top-12-free-open-source-tools-consider

Maturnuwun

@essanpupil