Speeding up Django / PostgreSQL

2014-08-20 (last modified: 2014-08-29)

Background

At my current contract, we process hundreds of thousand of articles a day and those are saved in some different models as we go along the pipeline. The stack for that application is Django (1.6) + PostgreSQL (9.3) + rq (alternative to the classic Celery for the queues).
Tasks started to take way too much time and so I started looking into.
This is not going to mention caching but things should obviously be cached if it is possible in your project.

Speeding up Django queries

The PostgreSQL database being on his own server, I assumed it was properly configured (spoiler: it wasn't) and thus started the investigation in the django project.
The models in that case are news articles, so it contains the text of the article (and it can get pretty big) as well as lots of metadata about it.
All in all, one of these objects can get pretty big (relatively speaking, keep in mind there are tens of millions of them) so getting/saving them can take some time.

Only

Most of the time you do not need to get the whole object, you might want to only get one or two fields, very simple thing to do in SQL and in Django as well thanks to .only():

```
# Equivalent to SELECT id FROM myapp_article;
Article.objects.all().only('id')
# Note that for .get(), only needs to be placed before .get() as only() is a metho
d
# of Queryset and .get() returns a model
Article.objects.only('id').get(id=42)
```

That sped up things in some places, but it was still pretty damn slow.

Update fields

Looking at the slow queries log (more on that in the postgres part below), the issue was very clear: when updating an article on one field, it was re-saving everything !

Django's ORM comes with an easy way to fix that:

```
# Equivalent to UPDATE myapp_article SET url='http://www.google.com' WHERE id = 4
2;
article = Article.objects.get(id=42)
article.url = 'http://www.google.com'
# article.save() -> naive way, will update every field
article.save(update_fields=['url']) # correct way
```

Bulk create

Another thing to try was to bulk insert articles instead of creating them while looping:

```
to_insert = []
for article in articles:
  if needs_saving:
    to_insert.append(article)  # article is a dict here

Article.objects.bulk_create([
    Article(**article) for article in to_insert
])
```

There are a few caveats to be aware of when using using bulk_create, those are explained in the django doc.

That's three problems fixed. Still slow though.

Mass update

In the same spirit of the bulk_create, mass update will also speed up your code quite a bit:

```
ids_to_update = []
# Code that appends to ids_to_update
Article.objects.filter(id__in=ids_to_update).update(failed=True)
```

This will be much faster than doing one update for each article individually as it will be done in a single SQL 'UPDATE' query.

Indices

Another one was to add an index on a column that was used to filter but wasn't an index at all:

```
external_id = models.IntegerField(db_index=True)
# And then run a south or django 1.7+ migration
```

Adding an index in postgres is pretty damn fast so do not be worried about the time it could take and sped up those queries significantly (ie they do not appear in the slow queries log).

Transactions

The last trick is to wrap a method/bit of code to ensure atomicity.
Django provides a method usable both as a decorator and as a context manager allowing to do that:

```
# As a decorator
@transaction.atomic
def do_lots_of_queries:
  pass

# As a context manager
with transaction.atomic():
  # do queries
  pass
```

This prevents from commiting every query and can speed up the piece of code significantly.
Use that sparingly though as transactions can have some performance cost, make sure it's actually better before putting everything in transactions.

After all these changes, the processing was faster but still super slow.
I decided to then look at the DB server but just an additional tip before.

Bonus: select_related and prefetch_related

Another thing I didn't use there (because in my case I didn't need to select related objects) is select_related and prefetch_related. Those are used when you need to get associated models you know you will need later in your code and don't want to do a query for each.
select_related is used where you have a one-to-many or one-to-one (and do a JOIN in SQL in one query) and prefetch_related for many-to-one and many-to-many (do one query per type of object and join them in python).
This can shave a huge amount of queries from your total number of queries: at my last job it went from ~1000 to ~50 just by using those.

For example, let's say you have an Article and Author model and you want to display a template with the article content and the author data.

```
# Let's assume there are 50 articles
articles = Article.objects.all().select_related('author')

{% for article in articles %}
  {{ article }} {{ article.author.name }}  # without select_related this will do a
n additional query per loop
{% endfor %}

# With select_related:  1 query
# Without: 51 queries
```

You can easily see from the example how it helps A LOT.
To easily spot the places where you can/should use them, install django-debug-toolbar and look at the SQL panels.

Postgres speed improvement

The first thing I did (and I actually did it before doing most changes I mentioned in the django part) is to activate the slow queries logging.
This will make postgres log everything that takes can more than X seconds (where you define X).
All the changes mentioned below are done in postgresql.conf, located in /etc/postgresql/9.3/main (change 9.3 for your version obviously).

```
# allows to log in the log directory specified below
logging_collector = on
# with that setting, logs will be saved in /var/lib/postgresql/9.3/main/pg_log/
log_directory = 'pg_log'
# Log queries that take more than 1s (I told you it was slow)
log_min_duration_statement = 1000
```

This allowed me to detect the slow queries caused by the absence of .only() and update_fields.

After doing all the fixes in the Django code I was still seing ridiculously slow queries (a good number were taking more than 20s).
Now when I added the slow queries logging, I noticed the absence of custom configuration at the bottom of the file.
Surely if there's a server dedicated to the DB, postgres will have been tuned to use all the

amazing amount of RAM available (I initially thought I had a bug when free -m told me the server had 64go of memory).
The tuning looks something like that (see the guide to know which values to choose):

```
log_min_duration_statement = 1000
max_connections = 100
shared_buffers = 15GB
effective_cache_size = 45GB
work_mem = 78643kB
maintenance_work_mem = 2GB
checkpoint_segments = 128
checkpoint_completion_target = 0.9
wal_buffers = 16MB
default_statistics_target = 500
```

This made everything run about 10x faster.

Conclusion

Most of the time, the biggest bottleneck will be the database so do not forget that you are not using SQL and therefore you might make mistakes using an ORM (forgetting the update_fields for example) that you would never do in SQL.
The first thing to check would be the postgres config if queries are running slow but don't forget to optimize the django part as well.
In this project, some queries will still require tuning through EXPLAIN (probably worth a different article). Including the postgres tuning and the django changes, the code is running 10 and 100x faster than it was before.