Speeding up a Django project

Paweł Marczewski
What this talk will be about

Django.

Some advice is Postgres-specific (but should be easy to adapt).

Mostly data-processing / database performance.

I hope for some advice from you as well!
Codility

A service for testing programming skills (think olympiad/contests, but with simple problems).

The website uses Django and PostgreSQL.

No strong backend/frontend divide (yet).

No huge amounts of data, but we’re running into performance problems from time to time.

Here’s what we came up with...
Use SQLite-in-memory for unit tests

DATABASES['default'] = {
    'ENGINE': 'django.db.backends.sqlite3',
    'NAME': ':memory:',
}

Blazing-fast startup time!
Use SQLite-in-memory for unit tests

It’s good to test on production engine as well (your CI server can do both).

Your code has to support SQLite.

Alternative (Postgres): turn off fsync for tests.
Other testing tips

If migrations are a bottleneck, you can squash them.

Parallelize your builds (useful if you have many Selenium tests).
Check your queries

django-debug-toolbar

https://github.com/django-debug-toolbar/django-debug-toolbar
Check your queries

Or, just look at plain Django logs (DEBUG level).
Use select_related

<table>
<thead>
<tr>
<th>ID</th>
<th>User</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="mailto:foo@example.com">foo@example.com</a></td>
<td>Foo</td>
</tr>
<tr>
<td>2</td>
<td><a href="mailto:foo2@example.com">foo2@example.com</a></td>
<td>Foo</td>
</tr>
<tr>
<td>3</td>
<td><a href="mailto:bar@example.com">bar@example.com</a></td>
<td>Bar</td>
</tr>
</tbody>
</table>
Use select_related

Rendering User.objects.all():

SELECT ... FROM users;
SELECT ... FROM accounts WHERE id = 1;
SELECT ... FROM accounts WHERE id = 2;
SELECT ... FROM accounts WHERE id = 3;
...

Use select_related

Rendering

User.objects.select_related('account'):

SELECT ... FROM users
    JOIN accounts
    ON users.id = accounts.user_id;

Much better!
Use prefetch_related

<table>
<thead>
<tr>
<th>ID</th>
<th>Account</th>
<th>Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Foo</td>
<td><a href="mailto:foo1@example.com">foo1@example.com</a>, <a href="mailto:foo2@example.com">foo2@example.com</a>, <a href="mailto:foo3@example.com">foo3@example.com</a></td>
</tr>
<tr>
<td>2</td>
<td>Bar</td>
<td><a href="mailto:bar1@example.com">bar1@example.com</a>, <a href="mailto:bar2@example.com">bar2@example.com</a></td>
</tr>
</tbody>
</table>
Use `prefetch_related`

Rendering `Account.objects.all()`:

```sql
SELECT ... FROM accounts;
SELECT ... FROM users WHERE account_id = 1;
SELECT ... FROM users WHERE account_id = 2;
SELECT ... FROM users WHERE account_id = 3;
...
Use prefetch_related

Rendering

Account.objects.prefetch_related('user_set'):

SELECT ... FROM accounts;
SELECT ... FROM users
    WHERE account_id in (1,2,3,4);
Or just drop to raw SQL

Blog.objects.extra(
    select={
        'entry_count': 'SELECT COUNT(*) FROM blog_entry WHERE blog_entry.blog_id = blog_blog.id'
    }
)
General rule:

Make $O(1)$ queries per page.
Do more in SQL than in your code

Example: data migrations
(convert data from one format to another).

Your database engine will be WAY more efficient at this than Python!

A complicated “UPDATE WHERE…” can be orders of magnitude faster than a for-loop.
Check what your queries are doing

Example from Postgres documentation.

EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

QUERY PLAN

Nested Loop (cost=4.65..118.62 rows=10 width=488)
  -> Bitmap Heap Scan on tenk1 t1 (cost=4.36..39.47 rows=10 width=244)
    Recheck Cond: (unique1 < 10)
      -> Bitmap Index Scan on tenk1_unique1 (cost=0.00..4.36 rows=10 width=0)
        Index Cond: (unique1 < 10)
  -> Index Scan using tenk2_unique2 on tenk2 t2 (cost=0.29..7.91 rows=1 width=244)
    Index Cond: (unique2 = t1.unique2)
Check what your queries are doing

In Postgres, EXPLAIN will give you a query plan.

EXPLAIN ANALYZE will also run the query and give you the timing.

Often, you’ll find out you need another index!

Even better: run statistics on the production database.

PostgreSQL 9.0 High Performance contains a good explanation of query plans and Postgres internals.
Memoize properties

class User(models.Model):

    @property
    def available_credits(self):
        return self.query_for_credits()
Memoize properties

@property
def available_credits(self):
    if not hasattr(self, '_available_credits'):
        self._available_credits = self.query_for_credits()
    return self._available_credits

Useful for complicated pages.

Downside: cache invalidation.
Use cache

Cache arbitrary data using cache.get() and cache.set().

Cache template fragments.

Use *django-cache-machine* to cache models (useful for data that changes rarely but is accessed often).
Compute things asynchronously

Example: our real time map widget.

Just now, a new test session started in United States

2,123,406 Assessments till date
Compute things asynchronously

Don’t do this:

```python
def get_map_data():
    if not cache.get('map_data'):
        data = compute_map_data()
        cache.set('map_data', data, 5 * 60)

    return cache.get('map_data')
```
Compute things asynchronously

Use a task queue, like Celery.

Support a “not ready yet” response and initiate re-computation.

Or just compute things periodically.

Good for pages with lots of views (like contest leaderboards).
Don’t auto-reload, poll

Our report used to auto-reload until it was assessed and ready.

However, polling a single AJAX endpoint (“is it ready yet?”) made the page less straining for our servers.
If all else fails...

Just use a stronger server!

c0d1l1ty

WE BOUGHT AN SSD
What comes next for us?

More aggressive HTML fragment caching.

Better frontend / backend split (serve static HTML and JS, pass data using JSON).

Sharding / horizontal database scaling.
Questions?
Comments?

You can reach me at pwmarcz@gmail.com
and http://pwmarcz.pl.